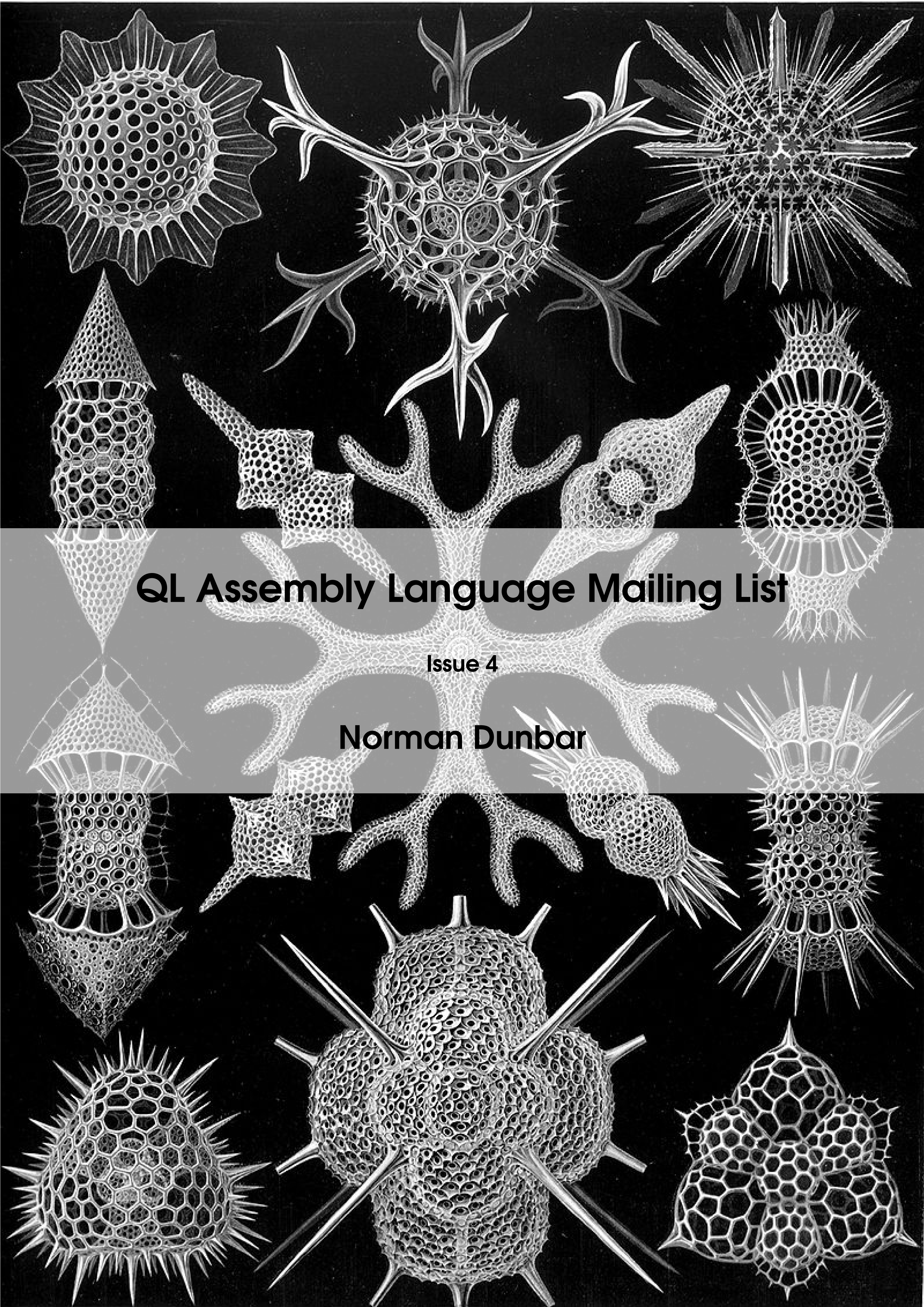
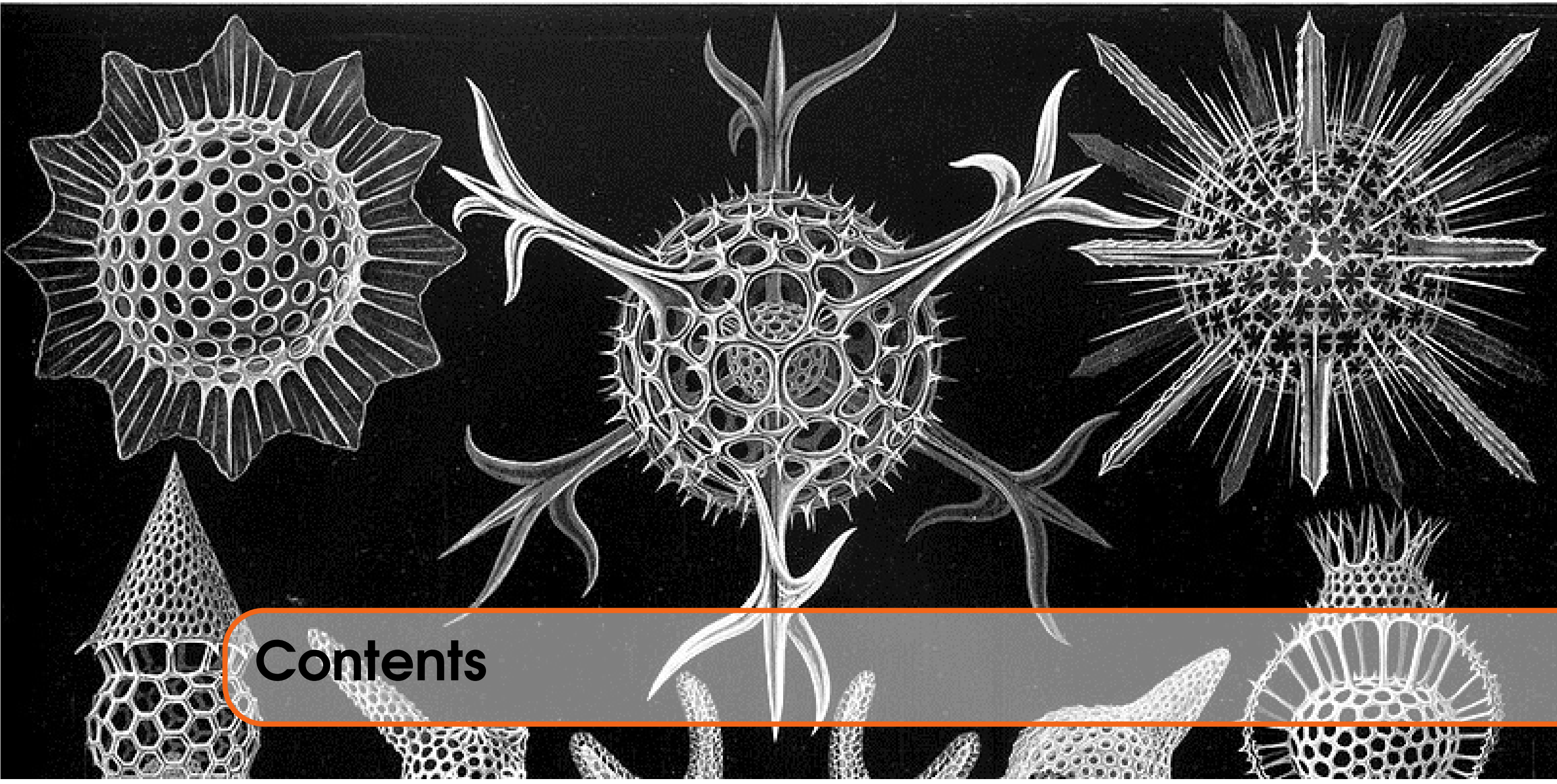


QL Assembly Language Mailing List

Issue 4

Norman Dunbar



Contents

Listings

1. Preface

1.1 Feedback

Please send all feedback to `assembly@qdosmsq.dunbar-it.co.uk`. You may also send articles to this address, however, please note that anything sent to this email address may be used in a future issue of the eMagazine. Please mark your email clearly if you do not wish this to happen.

This eMagazine is created in LATEXsource format, aka plain text with a few formatting commands thrown in for good measure, so I can cope with almost any format you might want to send me. As long as I can get plain text out of it, I can convert it to a suitable source format with reasonable ease.

I use a Linux system to generate this eMagazine so I can read most, if not all, Word or MS Office documents, Quill, Plain text, email etc formats. Text87 might be a problem though!

1.2 Subscribing to The Mailing List

This eMagazine is available by subscribing to the mailing list. You do this by sending your favourite browser to `http://qdosmsq.dunbar-it.co.uk/mailinglist` and clicking on the link "Subscribe to our Newsletters".

On the next screen, you are invited to enter your email address *twice*, and your name. If you wish to receive emails from the mailing list in HTML format then tick the box that offers you that option. Click the Subscribe button.

An email will be sent to you with a link that you must click on to confirm your subscription. Once done, that is all you need to do. The rest is up to me!

1.3 Contacting The Mailing List

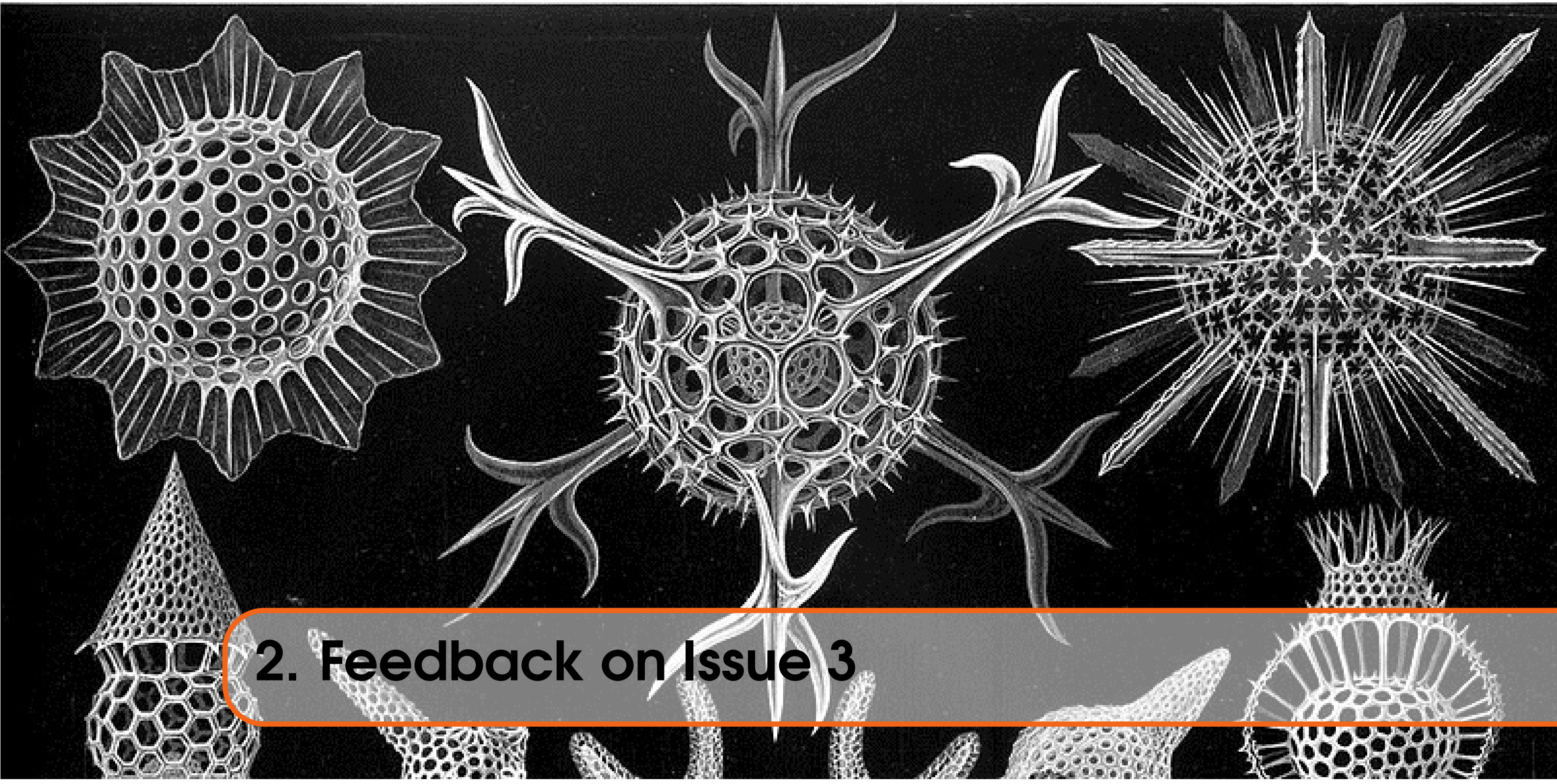
I'm rather hoping that this mailing list will not be a one-way affair, like QL Today appeared to be. I'm very open to suggestions, opinions, articles etc from my readers, otherwise how do I know what I'm doing is right or wrong?

I suspect George will continue to keep me correct on matters where I get stuff completely wrong, as before, and I know George did ask if the list would be contactable, so I've set up an email address for the list, so that you can make comments etc as you wish. The email address is:

`assembly@qdosmsq.dunbar-it.co.uk`

Any emails sent there will eventually find me. Please note, anything sent to that email address will be considered for publication, so I would appreciate your name at the very least if you intend to send something. If you do not wish your email to be considered for publication, please mark it clearly as such, thanks. I look forward to hearing from you all, from time to time.

If you do have an article to contribute, I'll happily accept it in almost any format - email, text, Word, Libre/Open Office odt, Quill, PC Quill, etc etc. Ideally, a LaTeXsource document is the best format, because I can simply include those directly, but I doubt I'll be getting many of those! But not to worry, if you have something, I'll hopefully manage to include it.



2. Feedback on Issue 3

I have received some feedback from a couple of my readers - Hi Wolfgang, Hi George - on a number of the topics covered in the last issue.

2.1 BubbleSort

2.1.1 Wolfgang Lenerz

WL: In line 76 of the bubble sort on page 17, you move the length of the string into d0. What happens if this is a 0 length string (could happen if one were to adopt your code to become a basic keyword, for example). In that case, lots of memory might be overwritten. Perhaps a `BEQ` to the end of the routine might be useful.

ND: Yes, good catch. I admit that I always do that! George has told me off for it many times. I though I'd learned but it appears not.

You are correct, if it was zero, it would result in lots of memory being sorted that perhaps didn't need to be!

This problem also appears at line 74 on page 16 too. The solution to both is simple, amend the code to amend *Listing 2.4 Bubblesort* on page 16 and *Listing 2.5 Better Bubblesort* on page 17 to the following, which is a corrected version on listing 2.5 on page 17 by the way.

```
;------------------------------------------------------------------------------
; ENTRY:
; For entry at label bubblesort:
;
; A1.L = Start address of data to be sorted. Word count first.
;
;------------------------------------------------------------------------------
; WORKING:
;
```

```
; A1.L = Start Address of data to be sorted, word count first.
; A2.L = Data being compared right now. (-1(a2) and (a2)).
; A3.L = Address of the Compare and swap routine.
; D0.W = 'n' = end of unsorted data.
; D1.B = Temp for swapping.
; D2.W = 'i' = loop counter.
; D3.W = 'newn' = last item sorted.
;-------------------------------------------------------------------
; EXIT:
;
; D0.L = 0.
; A1.L = Preserved - Start address of sorted bytes' word count.
; All other registers preserved.
;-------------------------------------------------------------------
bubblesort
        movem.l d1-d3/a1-a2,-(a7)
        move.w (a1)+,d0         ; N = length(a)
        beq.s bs_done           ; Nothing to do, bale out
        subq.w #1,d0            ; We need n-1 when testing
...
```

Listing 2.1: Better Bubblesort - Bug Fix 1a

And now we need a label bs_done at the end which will allow us to exit from the code if there are no items to be sorted.

```
bs_until
        bne.s bs_repeat         ; Until n = 0

bs_done
        movem.l (a7)+,d1-d3/a1-a2
        clr.l d0
        rts
```

Listing 2.2: Better Bubblesort - Bug Fix 1b

If you wish to save time very slightly, then you could rearrange the above code as follows, because D0 is already set to zero, there's no need for the register to be cleared:

```
bs_until
        bne.s bs_repeat         ; Until n = 0
        clr.l d0

bs_done
        movem.l (a7)+,d1-d3/a1-a2
        rts
```

Listing 2.3: Better Bubblesort - Better Bug Fix 1b

Of course, that's not all there is to it, as George points out below, what's the point of sorting zero or 1 items? You need at least two to get a decent sort.

2.1.2 George Gwilt

GG:I have two comments on the bubble sort, the first concerns what the program does and the second on how it does it.

2.1.3 Program Content

GG: The aim is to sort a set of integers into ascending numerical order. The list is scanned several times with successive pairs being swapped if needed to set them in the right order. Since each comparison takes place after any previous swap it implies that the largest item must inevitably end up at the end of the set after each scansion[1].

This means that the number of items scanned can be reduced by one at each go. The algorithm used performs all the scansions down to the last one of two items. One of the members of SQLUG suggested that if you detected a scansion during which there were no swaps you could stop the process at that point. This could reduce the time taken.

ND: I might not be reading the above correctly, but the algorithm used in my version does indeed stop when it reaches the last placed item as it “knows” that all following items are already sorted.

This is facilitated by comparing `D2.W` with `D0.W` at label `bs_end_if` where `D2.W` is the counter (N) through the array of bytes and `D0.W` is the index (NEWN) of the last item sorted on the previous pass.

Having said that, it is of course true that an early exit should be made when there were no swaps made in a pass, the array of bytes is sorted.

2.1.4 Program Coding

GG:A BASIC program expresses a FOR loop as proceeding from smaller to larger. For example:

```
FOR x = 1 to 9
```

Listing 2.4: SuberBasic FOR Statement

The literal translation of that to Assembly Language requires increasing x by one during each loop and comparing the new value of x with 9 to end the loop. However, the normal method of counting in Assembler programs is by the use of the DBcc instructions which combine the reduction of x from its top value to zero with the test.

2.1.5 A Suggested Alternative Program

GG: What follows is a subroutine which deals with both points. Four instructions are marked with asterisks. These instructions are the extra ones needed for earlier detection of completion. The instructions can be omitted if earlier detection is not wanted.

To detect if a swap has been made during the scanning of a list, the most significant bit of D2 is used. This is set to zero at the start of each scansion and is set to 1 whenever a swap occurs during that scansion.

This program allows for an error exit. This will occur if the word count of the string to be sorted is less than 2. You have to have at least two items to allow any comparisons. And what is the use of sorting a string length of one, or even zero?

ND: Yes, it’s obvious really isn’t it? Especially when someone points it out!

```
; At entry
;
```

[1] An interesting word that I had never seen or heard before, that I could recall. The definition is *The rhythm of a line of poetry, or the process of examining the rhythm of a line of poetry* from `https://dictionary.cambridge.org/dictionary/english/scansion`

```
;  A1.L = Start address of data to be sorted. Word count first.
;
; At exit
;
;  D0.L = error code
;  All other registers preserved



bs_reg     reg        d1-2/a1-2

bs_0       movem.l    bs_reg,-(sp)
           move.w     (a1)+,d2          number of items . .
           subq.w     #2,d2             . . less 2
           bpl        bs_1              OK
           moveq      #-15,d0           bad parameter . .
           bra        bs_5              . . error exit

bs_1       bclr       #31,d2            clear change marker  **
           move.w     d2,d0             length of list
           movea.l    a1,a2             point to start of items

bs_2       move.b     (a2)+,d1          current item . .
           cmp.b      (a2),d1           . . compare with next
           ble        bs_3              no change needed
           move.b     (a2),-1(a2)       swap . .
           move.b     d1,(a2)           . . items
           bset       #31,d2            mark change occurred  **

bs_3       dbf        d0,bs_2           count list
           tst.l      d2                any changes? . .      **
           bpl        bs_4              . . no - ended        **
           dbf        d2,bs_1           sort shorter string

bs_4       moveq      #0,d0             OK exit

bs_5       movem.l    (sp)+,bs_reg
           rts
```

Listing 2.5: Even Better Bubblesort!

2.2 Multiprint

2.2.1 Wolfgang Lenerz

WL: In the `Multiprint` routine on page 22, you have the `mp_loop` as follows:

```
mp_loop
    move.l a1,-(a7)            ; Save current string
    move.w ut_mtext,a2         ; Get the vector
    jsr (a2)                   ; Print current string
    bne.s mp_oops              ; Something bad happened
    move.l (a7)+,a1            ; Start of current string
    adda.w (a1),a1             ; Add size word
    addq.l #3,a1             ; Prepare to make even
    move.l a1,d5
    bclr #0,d5               ; D5 now points at next string
    move.l d5,a1             ; Back into A1
```

Listing 2.6: Original MultiPrint

I think this could be replaced by

```
mp_loop
    move.l a1,d5             ; Save current string
    adda.w (a1),d5           ; Preset pointer to next
    move.w ut_mtext,a2       ; Get the vector
    jsr (a2)                 ; Print current string
    bne.s mp_oops            ; Something bad happened
    addq.l #3,d5             ; Prepare to make even
    bclr #0,d5               ; D5 is now even
    move.l d5,a1             ; Back into A1
```

Listing 2.7: Wolfgang's MultiPrint

This is not only 2 instructions shorter, it also avoids using the stack, and thus accessing memory which is always slow.

ND: I like the stack! That's what it's there for. But yes, you are correct, I could have just used D5 in this manner and saved the need to push and pop pointer values to and from the stack, which is slow memory access rather than speedy register to register access.

How much speedier is using a register? Well, unfortunately, the more recent 680xx Programmer's[2] Manuals don't list the execution times of the instructions, but my old First Edition (1984) copy of *M68000 16/32-Bit Microprocessor Programmer's Reference Manual* lists them for the 68008 and the 68010. The following is from the 68008 section.

To stack `A1.L` using Address Register Indirect with Pre-Decrement requires a grand total of 24 clock cycles.

To simply move `A1.L` into `D5.L` takes a grand total of 8 clock cycles. So Wolfgang's method is quite a bit faster (ok, I admit it, it's 3 times faster for one instruction and I have two of these plus another two that Wolfgang has omitted, so even faster still!)

WL: I also have a question which might be due to the fact that I use Qmac and you use Gwasl which I don't know about.

You write your strings as follows

```
s1      dc.w s1e-s1-2
        dc.b 'This is a demo of MultiPrint '
s1e     equ *
        ds.w 0

s2      dc.w s2e-s2-2
```

Listing 2.8: MultiPrint String Table Example

We agree, I presume that if the length of the string is uneven, then label `s1e` points to an odd address. Is this why you use the `ds.w 0` afterwards, to make sure the following label at `s2` lies at an even address?

[2]Only *one* programmer?

Under QMAC this wouldn't be necessary since the label at `s2` starts with a `dc.w` and so Qmac would put it at an even address, avoiding the unnecessary `ds.w`, inserting a padding byte if necessary. I don't know whether this is a speciality of Qmac or if Gwasl does the same, in which case the `ds.w` would not be strictly necessary. (I know it doesn't actually add a word to the code itself).

ND: You are correct. It isn't really required as both Gwasl and Gwass automatically adjust the address to be even when a `dc.w` or `dc.l` directive is found. The various `ds.w 0` directives are not strictly necessary.

I did have to run a quick test with both of these assemblers to be absolutely certain though!

2.3 HexDump

2.3.1 Wolfgang Lenerz

WL: At label `hex_nibble` in the `hex_dump` program, instead of stacking `D4`, you could `move.b d4,d2` and do all the calculations on `D2`. No need, then, to get `D4` back from the stack.

ND: See above! I sit corrected. Thanks.

2.3.2 George Gwilt

GG: The Hexdump utility produces almost the same output as the menu item 7 of my program NET_PEEK so I was interested to see what were the differences.

The output is identical apart from the ASCII representation being enclosed in square brackets (NET_PEEK has none) and the unprintable characters are replaced by dot (full stop or period). The first of these is minor but the second, printing a dot for the unprintable, makes it impossible to tell which characters are really dots. I would regard that as a flaw, though a minor one.

ND: As I mentioned in the article, I am a frequent user of the Linux hexdump utility and decided that I needed something similar on my QL.

To this end, the utility as published prints the data in a manner pretty close to how running the command `hexdump -C some_file.bin` would. The only difference is that where I use square brackets, hexdump uses pipe characters, aka '|' instead.

Hexdump also displays unprintable characters as dots, but I wonder how NET_PEEK displays them? I've never used NET_PEEK - possibly because the name implies something to do with the QL's network, which I've pretty much never used.

Does NET_PEEK display unprintable characters with a space? Or a question mark? Whatever character is being printed will cause confusion as there will be a printable character representing an unprintable one!

Still, as George says, it's a minor flaw, however, it's pretty much a standard to print a dot - I've seen it on various utilities in Linux, Unix, ICL VME mainframes, IBM mainframes and even, Windows.

GG: The program differs in the method of producing the HEX. Although NET_PEEK has been available for some time now and is expected to work on all types of QL it uses the QDOS ITOH hex routines and these seem to be OK.

ND: I have a vague recollection of checking the QDOS version with the `MT_INF` trap call and if it was 1.03 and above, I used the internal routines but if not, I used my own. Eventually though, I gave up on having two ways to do the same thing and simply used my own for everyone.

I checked the various docs.

Note

Read on and see just how easily I get confused. I'm about to embarrass myself by getting completely the wrong end of the stick, plus, I've been doing it for years! Maybe I should just give up now while the going is good?

The penny finally dropped just after I read the docs in Tebby & Karlin.

I could have removed all this, but I'm leaving it in as a reminder that some routines don't work, but mainly as a reminder to myself to *pay attention and learn to read properly!*

Pennell states in *The Sinclair QDOS Companion* on page 110 that *there are many QDOS vectors to convert between various bases, but unfortunately several do not work in current versions of QDOS.* Dickens on the other hand, is more specific in his *QL Advanced User Guide* where he states that it is version 1.03 of QDOS, and below, where the conversion routines do not work, and specifically notes this against the following on pages 234 onwards:

- `CN_BTOIB` $104 ASCII Binary to byte.
- `CN_BTOIW` $106 ASCII Binary to word.
- `CN_BTOIL` $108 ASCII Binary to long.
- `CN_HTOIB` $10A ASCII Hex to byte.
- `CN_HTOIW` $10C ASCII Hex to word.
- `CN_HTOIL` $10E ASCII Hex to long.

Pennell also notes that these are broken on page 112. Apparently they *contain a large number of mistakes, and so do not work.* :-(

I've used my own versions for many years in numerous (well, quite a few) of my programs where it was required to avoid those bugs in QDOS versions from causing hard to track down problems if anyone using older (JM and previous?) ROMs. I know that they work on JS ROMs as my copy of Pennell is annotated as such for each of the above conversion routines.

I wonder if anyone still uses JM based QLs?

Tebby and Karlin in *QL Technical Guide* also mention that the above are fubar[3] in QDOS 1.03 and below.

Of course, if I had been paying attention, I'd have noticed that these broken routines are the ones which convert *from* an ASCII string of binary or hex digits *to* a value on the maths stack. These are not what I needed to use, so I checked the docs again. The various conversion *from* binary *to* ASCII strings routines *do* work, and have done in all versions of QDOS it seems! One day, I'll learn to read[4]. Sigh.

2.4 Jump tables

2.4.1 George Gwilt

GG: Curiously enough NET_PEEK, as well as producing a hex dump, also makes use of a Jump Table.

The table is exactly of the format described in Assembly_Language_003 but the coding differs slightly, partly because there seems to have been an instruction left out in `got_good_option`, the last three instructions of which are:

[3] FUBAR - F*d Up Beyond All Recognition!

[4] Somehow I doubt it!

```
            lsl.w       #1,d0
            lea         JumpTable,a2
            jsr         (a2,d0.w)
```

Listing 2.9: Jump Table Code Extract

(This is from page 39.)

The contents of `(a2,d0.w)` are the offset from JumpTable to the option denoted by the content of `D0.W`. Alas, the actual program resides at the address whose value is the offset plus the address of the jump table.

Here is what NET_PEEK does at this point in its program:

```
            add.w   d7,d7           (a differed doubling and of D7 not D0)
            lea     prog,a2         The jump table
            move.w  (a2,d7.w),d7    Set D7 to the offset from the Table
            jmp     (a2,d7.w)       Jump to the program
```

Listing 2.10: NET_PEEK Code Extract

I think that the instruction `move.w (a2,d0.w),d0` must have been forgotten in the printing of Jump Tables.

ND: Yes, this is indeed a bug, which is strange as it's extracted from a working program. I'll need to check that now!

Ok, I checked. The working program does have the missing instruction present, so it's a copy & paste error on my part. It also uses `JMP`.

GG: It is a very minor point, but the doubling of `D0` is carried out using `LSL` whereas I have used `ADD` to double `D7`.

Is there any reason for preferring the use of `LSL` rather than `ADD` to double a number?

ND: Timings perhaps? Because I've always done it that way? Someone told me it was quicker? I checked the 68008 timings for `add.w d0,d0` and `lsl.w #1,d0`. The former takes 8 cycles while the latter takes 12. Looks like I've been misinformed again! `ADD` is about 30% quicker.

GG: My reason is purely personal. I would always use either `LSL` or `ASL` to raise to a power of two greater than one. However, each time I have to decide which to use. Does it make a difference? If so what? Also I have to remember to use `LSL` (or `ASL`) instead of `LSR` (or `ASR`). I determine this always by pointing to the left (or right) and then deducing whether the shift will make the number bigger or smaller. The use of `ADD` is thus, for me, much simpler!

ND: Phew! I see what you mean, logical shifts or arithmetic ones? Who needs them when we can simply double a number by adding it to itself!

The ARM processor doesn't have, as far as I remember[5], an instruction to shift (logical or arithmetical) one place left or right. You can use the instruction in your source code, but either the assembler converts it to an `ADD` instruction, or, the bit pattern in the assembled binary code is exactly the same as that for an `ADD` - it's one or the other. (Apologies in advance if I've got it wrong here, my ARM docs are not at hand!)

GG: The reason for accessing the required program by `JSR` rather than `JMP` is given on page 36. Since NET_PEEK uses `JMP` I wondered if I had missed out here. However I could not easily see how I could rewrite the code with `JSR`.

[5]I might just be thinking of Atmel AVR assembly language here actually, an 8 bit microcontroller forund in the Arduino, for example.

I wondered how a very much simplified system would be improved by the use of `JSR`. Let us suppose that there are four steps in such a program as follows:

1. Display Options.
2. Find the Choice.
3. Execute the chosen option.
4. Return to 1.

If we use `JSR` in step 3 then each option program must end `RTS` and step 4 becomes, in the main program, the branch instruction to step 1.

If we use `JMP` in step 3 instead of `JSR` then each option program must end with a branch to step 1 and there is no such branch in the main program.

Thus using `JMP` instead of `JSR` reduces the program by one instruction - a good result, both in terms of length and time.

ND: Yes, timings. Yet again, your version outperforms mine. `JMP (a2,d7.w)` takes 22 cycles plus a further 18 for the `BRA` to exit each routine. `JSR (a4,d7.w)` requires 38 plus 32 to execute the `RTS` and yet another 18 for the `BRA` that comprises step 4 above.

GG: What am I missing?

I examined NET_PEEK more closely and found that the final branches of some options entered at the last few instructions of some other option. There was no set of instructions common to all options which could have been put in the main program and obeyed after `RTS` from every option. So `JMP` seems to be necessary for my NET_PEEK.

ND: Technically, if I'm reading the above correctly, your routines could have been called with `JSR` and ended with an `RTS` instruction where necessary. Those routines that exit via the last few instructions of another routine could have continued to do so. You could therefore have used `JSR` instead of `JMP` and had the same effect, only slower!

A Further Twist

GG: Each line in the Jump Table ends " - JumpTable". When there are many items in a table it can become tedious to type these last few characters each time.

ND: True, at least using a QL to do the source code editing. I'm afraid that I have found that after too many years using Windows and Linux editors, that they are pretty much standard in the use of editing keys etc.

Also, and most irritatingly, when I use CTRL+ALT+LEFT or CTRL+ALT+RIGHT to delete to the start or end of a line, in Linux it switches my desktops around because Linux grabs the key strokes before QPC gets them!

For those who still use Windows, you have a single desktop. In Linux, I have as many as I like, each with different icons on the desktop, programs open in them etc. I can have my source editing on desktop 1, assembler and QPC on desktop 2 and so on. I can switch between them at will.

I use a Linux editor to write my source code these days, and simply open it in QPC, assemble it, fix it in QED while still in QPC. Eventually I get a working version which I save the updated source code back to Linux for inclusion in the articles.

I use copy and paste to generate all those nasty repetitive bits. Or, indeed, sometimes I can use a macro command in the Linux editor.

GG: A very slight change to the coding can reduce the typing. In this version each line of the table contains the offset, not from the start of the table but from that very line. This is achieved by using asterisk instead of the table's name. Thus each line of the table would end " - *"

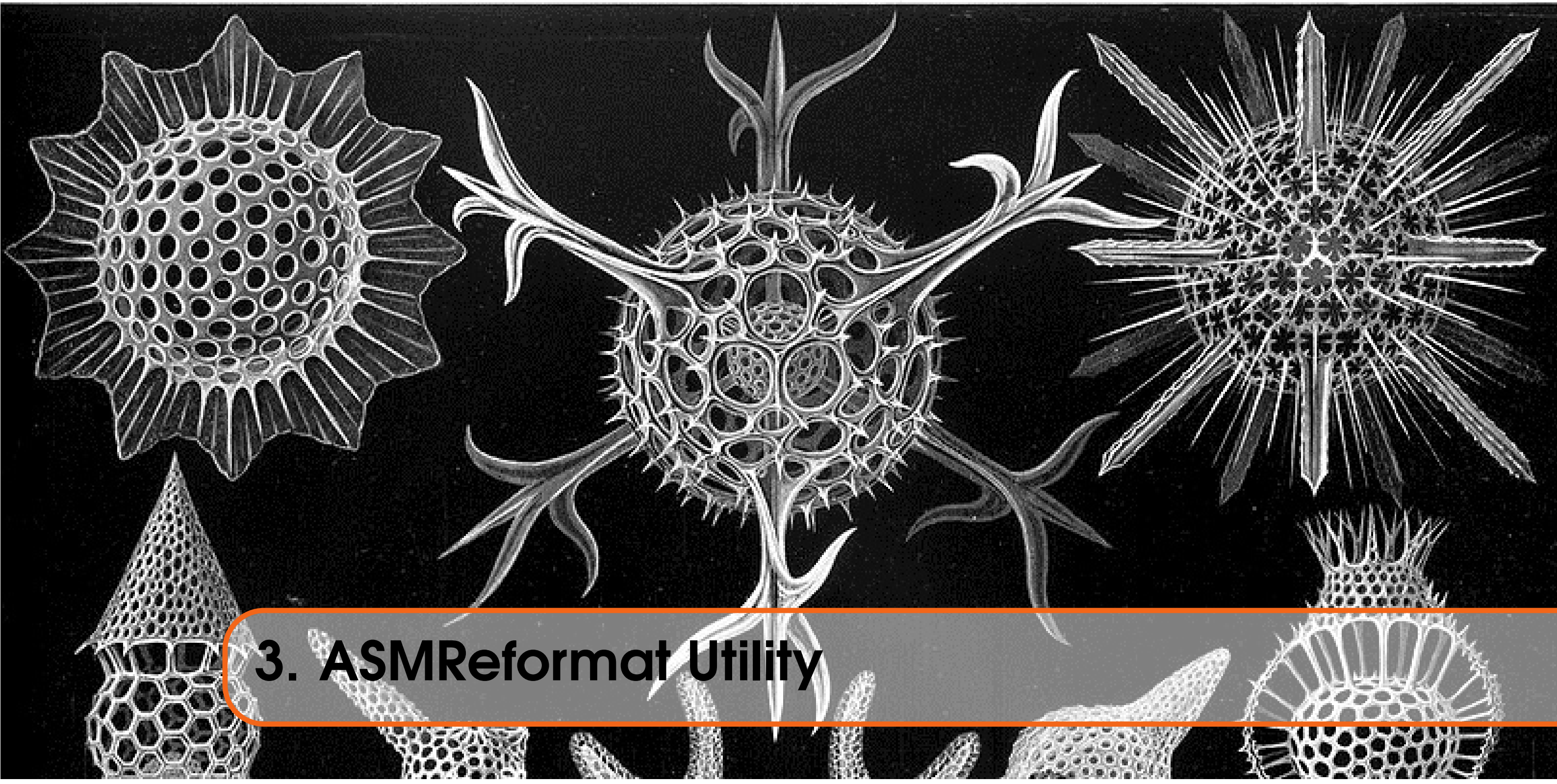
The instructions could now be:

```
            lea         JumpTable(d0.w),a2
            move.w      (a2),d0
            jmp         (a2,d0.w)
```

Listing 2.11: Improved Jump Table Code

The first instruction sets `A2` pointing to the appropriate line in the table, not its head.

ND: Saves typing and reduces the program by another instruction too.



3. ASMReformat Utility

ASMReformat is a utility I wrote out of necessity. When I decided to upload DJToolkit to Github, I needed to convert all the tab characters into spaces (4 per tab) and to reformat the code so that it was consistent - labels in one column, opcodes in another column, operands in another and comments somewhere else on the line and so on.

I started doing this manually but gave up after a while as it was tediously boring. As I was at work, where I used to do QL stuff in my lunch hour, I decided to write a program to assist in the onerous task of reformatting my code into something acceptable. Apart from personal desires, if the code is on Github, then anyone who wants to can download it and amend it as they see fit. With the original code, there were tab characters all over the place and in my setup, these set the positions on the line to 12, 20 and 40 (operands, opcodes and comments) which the Editor SE allowed me to do. Nowadays, I can find very few editors that allow asymmetric tabs in this manner, so opening `DJToolkit_asm` in a normal editor would have messed up the code. So, tabs needed to be converted to spaces. Of course, with only three tabs, maximum, per line, setting tabs to any given width was going to be fraught! Some of my comments were a little all over the place too, but that's a smaller problem.

Given that I was at work, the options open to me were Java[1], C or C++ as Oracle PL/SQL wouldn't really have cut the mustard! C++ it was then.

If you look on github (`https://github.com/SinclairQL/DJToolkit`) you will notice a file named `DJTKReformat.cpp` in the `tools` folder. That's the utility I wrote to do the reformatting and while it worked fine, I've since discovered a bug in that it doesn't correctly format lines where a literal, or value, continues over more than one line, for example:

```
        ...
Message dc.w MessageEnd-Message-2        ; Word count
        dc.b 'Some text goes here.',     ; To be continued...
        dc.b ' And is continued here',
        dc.b linefeed
```

[1]Which I loathe and detest with a vengeance!

```
MessageEnd  equ  *
        ...
```

Listing 3.1: Continuation of Operands

A minor problem as it doesn't prevent the reformatted code from assembling, it just doesn't look nice - to my mind anyway. Plus, I'm a little worried that a utility I write, which might be useful to others, is written in such a way that you need a PC or a MAC to be able to reformat the source code of a QL assembly language program. The only solution was a complete rewrite in assembly.

3.0.1 Settings

The program is hard coded with the following settings:

- Labels start in column 1 (or zero if that's what your text editor says - QD, I'm looking at you!)
- Opcodes, ie the actual instructions, start in column 12. (Aka column 11 in QD.)
- Operands, for those instructions which need them, start in column 20. (Or column 19 in QD etc.)
- Comments, in line, start in column 40, *unless*, the operand has exceeded column 38, in which case there will be a pair of spaces after the operand and the comment will begin in the next column after those two spaces. Adjust the column numbers to suit QD as above!
- Full line comments always start with a semicolon (;), or asterisk (*) in the first column.
- Blank lines are either lines where the only character is a linefeed, or, lines where there are no characters other than spaces, tabs and the final linefeed. This latter option is one which the original C++ utility did not handle specifically, but the library code used did it internally. This caused me some serious bug hunting as QD seems to have a nasty habit of silently inserting tab characters when I don't need or want them.
- When parsing the original source code lines for labels, opcodes, operands and comments, the linefeed indicates the end of the input, and spaces or tabs (or the linefeed of course) indicate the end of whatever I'm parsing at the time.
- Comments are allowed spaces and tabs. Labels, opcodes are not. Operands *may* allow spaces and tabs, but only if they are wrapped in quotes - single or double.

If you do not wish to use my settings, simply locate the equates in the source code for the 4 column places and adjust to suit your wish. I'm pretty sure that labels, in all assembler rules, must begin in the first column of the line, where they are present.

```
; Where the text goes on the output line(s). We need to offset
; these by -1 as we count from zero in the buffers.
labelPos
           equ     1-1                 ; Labels in column 1
opcodePos
           equ     12-1                ; Opcodes in column 12
operandPos
           equ     20-1                ; Operands in column 20
commentPos
           equ     40-1                ; Comments in column 40
```

Listing 3.2: Configuration of Column Positions

3.0.2 Usage of ASMReformat

As with many of my recent articles[2], `ASMReformat` is written as a filter, so execution is as follows:

```
ew ASMReformat_bin , 'input_file ', 'output_file '
```

Listing 3.3: Executing ASMReformat

You will notice that I've used `EW` rather than `EX` as this returns an error code (and message) if there are problems with the execution. If you use `EX` instead, you don't see any error messages if the utility encounters any errors in its execution.

The basic steps carried out by the utility are as follows:

- Check the correct number of channels have been supplied.
- Reads the input file in a loop, until EOF is detected. If errors occur, exit the utility.
- For each line of input read, the following processing takes place:
  - If this is an operand continuation line then
    - Extract the operand continuation.
    - Extract any comments, if present.
    - Format the operand & comments.
    - Write the reformatted line to the output, followed by a newline.
    - Skip back to the main loop again.
  - If this line is entirely a comment, write it to the output file unchanged, and skip to the main loop start again.
  - If the line is blank, or has no actual content, write a blank line to the output file and skip back to the main loop start again.
  - Extract the label, if present.
  - Extract the opcode (or instruction) if present.
  - Determine if the opcode needs an operand to be presentt.
  - Extract the operand, if one is necessary.
  - Extract any comments at the end of the line.
  - Reformat the line as required.
- The reformatted line is written to the output file, followed by a linefeed.
- The main loop is then executed again.
- At the end of file, the utility exits with no errors.

3.0.3 Field Extraction

To extract any of the required fields from the source line, the utility will first scan from the current position on the line until it hits a character that is not a space or tab. Actually, it scans for anything that is higher in the ASCII table than a space. A linefeed or character 10 ($\$0A_{hex}$) indicates the end of the extraction.

During extraction, all characters are read and buffered until a space, tab or linefeed[3] is detected in the line of text from the input file.

For labels and opcodes, the text is simply extracted until a terminating character is read. For operands we have to be careful that we don't prematurely end the extraction during the parsing of a quoted string - all characters within the string are valid. Finally, comments allow all characters up to the terminating linefeed.

[2] Yes, I know, it's been a long while since the last edition. Sorry about that, but I've been busy!

[3] Once again, it's actually any character with an ASCII code less than or equal to a space - it's easier that way!

3.1 The Source Code

Having explained briefly what is happening, I'll now dive into the source code and attempt to explain what is happening.

Note By the way, I will soon have the code uploaded to Github to save you all having to type in the long listings. When I get a chance, I'll upload all the code from previous articles too.

```
;------------------------------------------------------------------------------
; ASMReformat:
;
; A filter program using an input and output channel, passed on
; the stack for its files.
;
; EX ASMReformat_bin, input_file, output_file_or_channel
;
;------------------------------------------------------------------------------
; 29/12/2017 NDunbar Created for QDOSMSQ Assembly Mailing List
;------------------------------------------------------------------------------
; (c) Norman Dunbar, 2017. Permission granted for unlimited use
; or abuse, without attribution being required. Just enjoy!
;------------------------------------------------------------------------------

me
            equ     -1                      ; This job
infinite
            equ     -1                      ; For timeouts
err_bp
            equ     -15                     ; Bad parameter error
err_ef
            equ     -10                     ; End of file


; Flag bits in D5.B:
inComment
            equ     0                       ; No comments on this line
noOperand
            equ     1                       ; This opcode has no operands
continue
            equ     2                       ; Operand continues on next line
lfRequired
            equ     3                       ; Do I need to print a linefeed?

; Resets inComment, noOperand and lfRequired flags
flagMask
            equ     %11110100               ; Reset flags
lowerCase2
            equ     $2020                   ; Mask to lowercase 2 characters
lowerCase1
            equ     $20                     ; Mask to lowercase 1 characters


; Various character constants.
linefeed
```

```
                equ     $0A                         ; You can probably guess these!
space
                equ     $20
comma
                equ     ','
tab
                equ     $09
semiColon
                equ     ';'
asterisk
                equ     '*'
dQuote
                equ     '"'
sQuote
                equ     "'"
```

Listing 3.4: ASMReformat Source - Equates etc

The first part of the listing is nothing exciting I'm afraid, it consists of a number of equates. Moving on...

```
; Where the text goes on the output line(s). We need to offset
; these by -1 as we count from zero in the buffers.
labelPos
                equ     1-1                         ; Labels in column 1
opcodePos
                equ     12-1                        ; Opcodes in column 12
operandPos
                equ     20-1                        ; Operands in column 20
commentPos
                equ     40-1                        ; Comments in column 40
```

Listing 3.5: ASMReformat Source - Configuration Section

This is the section that allows you to reconfigure my default options to suit your code writing style. Remember to subtract 1 from each "tab" position that you wish to use - offsets into the buffer used to reformat the lines of code are indexed from zero. Some editors, notably `QD` number column positions from zero, while others do it from 1. Strangely enough, `QD` numbers lines from 1 - hmmm!

```


; Stack stuff.
sourceId
                equ     $02                         ; Offset(A7) to input file id
destId
                equ     $06                         ; Offset(A7) to output file id
paramSize
                equ     $0A                         ; Offset(A7) to command size
paramStr
                equ     $0C                         ; Offset(A7) to command bytes

;=======================================================================
; Here begins the code.
;-----------------------------------------------------------------------
; Stack on entry:
;
; $0c(a7) = bytes of parameter + padding, if odd length.
; $0a(a7) = Parameter size word.
; $06(a7) = Output file channel id.
; $02(a7) = Source file channel id.
; $00(a7) = How many channels? Should be $02.
;====================================================================
```

Listing 3.6: ASMReformat Source - Stack Offsets

The code above defines the offsets onto the stack where the utility will find the count of files that should be on the stack - we need two of those, and where the two file IDs will be found when we need them.

The size and contents of the command string passed are also defined here, but currently, there are no uses for a command string for this utility and they are simply ignored. If necessary, and for a bit of homework, you *could* amend the program to accept a command line consisting of 4 numbers, comma separated, that define the desired "tab" positions for the output. It's up to you!

```
start
            bra.s       checkStack
            dc.l        $00
            dc.w        $4afb
name
            dc.w        name_end-name-2
            dc.b        'ASMReformat'
name_end
            equ         *

version
            dc.w        vers_end-version-2
            dc.b        'Version 1.00'
vers_end
            equ         *


bad_parameter
            moveq       #err_bp,d0              ; Guess!
            bra         errorExit               ; Die horribly
```

Listing 3.7: ASMReformat Source - Start Here!

Finally, we get to some actual code. The section above consists of the standard QDOSMSQ job header and some error handling for those occasions when we get too few or too many files on the stack at runtime.

```

clearBuffers
            lea         labelBuffer,a0
            clr.w       (a0)                    ; Nothing in labelBuffer
            lea         opcodeBuffer,a0
            clr.w       (a0)                    ; Nothing in opcodeBuffer
            lea         operandBuffer,a0
            clr.w       (a0)                    ; Nothing in operandBuffer
            lea         commentBuffer,a0
            clr.w       (a0)                    ; Nothing in commentBuffer
            rts
```

Listing 3.8: ASMReformat Source - Clear Buffers

The subroutine above is called from the main loop to make sure that the 4 buffers used for the 4 different fields of a source code line, are empty before we read the next line from the input file.

You can see that I've not bothered space filling the buffers - which would slow down the processing of a file - I simply set the word count to zero. When fields are extract from a source line, the appropriate word counts are correctly set so there are no spurious characters left over from previous lines to worry about.

```

;-----------------------------------------------------------------------
; Check the stack on entry. We only require two channels and if a
; command string is passed, we simply ignore it - for now anyway!
; We initialise the flags in D5.B to all off.
;-----------------------------------------------------------------------
checkStack
            cmpi.w   #$02,(a7)            ; Two channels is a must
            bne.s    bad_parameter        ; Oops
            moveq    #0,d5                ; Clear all flags
```

Listing 3.9: ASMReformat Source - Check Stack

A simple check of the number of opened files is done first. If we have two file IDs then we are good to go, otherwise, we bale out with a bad parameter error.

```

startLoop
            moveq    #infinite,d3         ; Timeout - preserved throughout

;-----------------------------------------------------------------------
; Clear all the buffers and set up for the next read of the input
; file. On EOF, we are done here, on error, we exit. This will return
; the error code to SuperBASIC only if we EXEC_W/EW the program, EX
; will never show the error.
;-----------------------------------------------------------------------
readLoop
            andi.b   #flagMask,d5         ; Reset some flags
            bsr.s    clearBuffers         ; Clear all buffers
            move.l   sourceId(a7),a0      ; Input channel id
            lea      inputBuffer+2,a1     ; Buffer for read to use
            move.l   a1,a4                ; inputPointer for later
            moveq    #io_fline,d0         ; Fetch a line and LF
            move.w   #1024,d2             ; Maximum buffer size = 1024
            trap     #3                   ; Read next line
            tst.l    d0                   ; Did it work?
            beq.s    checkContinue        ; Not EOF yet, carry on
            cmpi.l   #err_ef,d0           ; EOF?
            beq      allDone              ; No, exit the main loop
            bra      errorExit            ; Something bad happened then
```

Listing 3.10: ASMReformat Source - Main Loop

Here we begin the main loop. First of all, and just before we start it, we set an infinite timeout in `D3`. We only have to do this once as that register is preserved throughout all the calls we make to QDOSMSQ.

The loop begins by clearing out some flags that may have been set on the previous pass. These flags indicate the there was a comment on the previous line, that the previous line's opcode did not require an operand and the previous line required to be followed by a linefeed. We do not clear the continuation flag as we may still be processing a continuation line.

The code reads up to 1024 characters, including the linefeed, from the input file. Any errors, other than EOF, cause the utility to abort, hopefully returning the error code to SuperBASIC as it dies horribly! At EOF, the utility exits quietly and without any fuss.

```

;------------------------------------------------------------------------
; The read was ok, so we need to check if this line is a continuation
; of the operand from the previous line. We also store the word count
; of the string just read at the start of the input buffer.
;------------------------------------------------------------------------
checkContinue
            move.w  d1,-2(a4)           ; Save the string size
            btst    #continue,d5        ; Continuation set?
            beq.s   checkAllComment     ; No, skip

;------------------------------------------------------------------------
; We are on a continuation line, so extract the operand and that will
; also reset/set the continuation flag if necessary for a further
; continuation of the operand.
;------------------------------------------------------------------------
doContinue
            bclr    #lfRequired,d5      ; Nothing printed yet
            bsr     extractOperand      ; Extract operand and set flag
            bsr     extractComment      ; Grab any comments as well
            bsr     clearBuffer         ; We always do this here
            move.l  destID(a7),a0       ; Output channel Id
            bra     doOperand           ; And continue from there
```

Listing 3.11: ASMReformat Source - Operand Continuations

The code above processes any operand continuation lines. These are lines such as the following example:

```
            ...
Message     dc.w MessageEnd-Message-2
            dc.b 'This is the start of the text, ',  ; To be continued
    ⟹  ...
            dc.b 'and this is the end.',
            dc.b linefeed
            ...
```

Listing 3.12: Example Operand Continuation

As you see there some instructions (or assembler directives) can have their operand split over many lines by the use of a trailing comma (before any comments of course).

The section of code under discussion checks the flag and if set, the previous line was part of a continuation. In this case, we call the subroutines that extract an operand and any following comments, clear the input buffer - which is now being used as the output buffer, grab the output file ID and branch to the code which writes out the operand followed by the comments, if present. From there, the code will return to the start of the main loop to process the next line of input, which may also be a continuation.

```

;------------------------------------------------------------------------
```

```
; Is this line completely a comment line - in other words, is the
; first character a '*' or a ';' which indicates that it is a comment
; line. Write it to the output, unchanged, if so, then read the next
; line.
;------------------------------------------------------------------------------
checkAllComment
            cmpi.b  #semiColon,(a4)     ; Comment flag?
            beq.s   doWriteComment      ; Yes
            cmpi.b  #asterisk,(a4)      ; Or a comment flag?
            bne.s   checkBlank          ; Not this kind, no.

doWriteComment
            move.l  destID(a7),a0       ; Output channel Id
            lea     inputBuffer,a1      ; Buffer to write
            bsr     doWrite             ; Write out a line
            bra.s   readLoop            ; And go around again
```

Listing 3.13: ASMReformat Source - Comment Lines

If this line of source code is not a continuation, then the code above attempts to find out if it is a single line comment. Quite simply, if there is a semicolon (;) or an asterisk (*) in the first column, it's a comment line and if so, we simply write it to the output channel unchanged, then skip back to the top of the main loop.

```

;------------------------------------------------------------------------------
; If D1.W = 1 we must assume it is a linefeed only, so this line is
; blank. In this case we simply write it out.
;------------------------------------------------------------------------------
checkBlank
            cmpi.w  #1,d1               ; Linefeed only read in?
            beq.s   doWriteComment      ; Print out blank line.
```

Listing 3.14: ASMReformat Source - Checking for Blank Lines

Assuming that the source line read in is not a comment line, is it a completely blank line? These are easily determined as the line length put into `D1.W` by the read routine includes the linefeed. If the counter happens to be 1, then we must only have read a linefeed character.

If this is the case, we skip off to `doWriteComment` where we simply write out the input buffer to the output then return to the top of the main loop. This effectively writes a linefeed to the output file - as we desire.

```

;------------------------------------------------------------------------------
; Does the line actually have any content, if not just a linefeed?
; Can you tell I got trapped in this? A4 = first character of the
; input buffer, just after the word count.
; Calling scanForward adjusts A4 to the first non tab/space character
; in the input buffer. If A4 points at a linefeed, the line is blank.
;------------------------------------------------------------------------------
checkContent
            bsr     scanForward         ; Return A4 at first character
            cmpi.b  #linefeed,(a4)      ; Is line blank?
            bne.s   gotContent          ; We have content - extract it

;------------------------------------------------------------------------------
```

```
223 ; We have hit the linefeed, so there's no actual content on this line
224 ; only tabs and/or spaces. Print a blank line to the output.
225 ;------------------------------------------------------------------------
226 gotNoContent
227             move.l  destId(a7),a0       ; Output channel ID
228             bsr     doLineFeed          ; Print a linefeed
229             bra     readLoop            ; Go around
230 
231 ;------------------------------------------------------------------------
232 ; We do have content, so go process it.
233 ;------------------------------------------------------------------------
234 gotContent
235             lea     inputBuffer+2,a4    ; Reset input pointer
236             bra     extractData         ; No, do the necessary
```

Listing 3.15: ASMReformat Source - Checking for Content

Originally, I *thought* that the above check for blank lines would suffice, after all, it's how I make a blank line - simply press the ENTER key. However, QD seems to have other ideas and I spent a lot of time hunting down a nasty bug whereby the output file was *all over the place*.[4]

I eventually discovered the problem by running the `HexDump` utility from a previous issue of this somewhat irregular eComic, and that showed that the blank lines in question had spaces, tabs and other invisible character and did not consist of a single linefeed after all. Sigh.

The code above was written so that whenever a line is not blank, or suspected of not being blank, it will be scanned to see if there are characters that make it a valid source line.

The code calls `scanForward` to do this and the character pointed to by `A4` on return is either a linefeed for the end of the input text, or a character which is above the space in the ASCII charts - a printable character in other words.

If we have a linefeed, we skip off to write a linefeed to the output and rejoin the main loop at the top again, otherwise, we have got some content on the line and must process it. That processing starts a little way down the listing, so we skip over the following subroutines - which are involved in the extraction of the various fields in a source code line - to the code at label `extractData` which we can find at line 493 below. Did I mention that there are a few subroutines coming up next?

```
237 
238 ;------------------------------------------------------------------------
239 ; Copy any label from the inputBuffer to the labelBuffer. A4 is the
240 ; input buffer and we should be sitting at the start.
241 ; We assume there will be no label - most assembly lines have no
242 ; label - and check from there. A label has a non-space/tab/newline
243 ; in the first character, anything else is assumed to be a label. As
244 ; all those non-label characters are less than a space (ASCII) then
245 ; a simple test for anything lower or eqal to a space is done.
246 ;------------------------------------------------------------------------
247 extractLabel
248             cmpi.b  #space,(a4)         ; First character a space?
249             bls.s   extractLabelDone    ; Yes, exit - no label
250 
251 ;------------------------------------------------------------------------
252 ; We have a label, copy it to the labelBuffer. Keep a count of chars
253 ; copied in D0.
```

[4]A technical term!

```
;------------------------------------------------------------------------------
            lea     labelBuffer+2,a5    ; Our output buffer

doCopyText
            move.l  a5,a1               ; Save buffer
            bsr     copyText            ; Go copy it

extractLabelDone
            rts
```

Listing 3.16: ASMReformat Source - Extracting Labels

The subroutine above extracts labels from the start of a line. If the character at the start of the line is higher up the ASCII charts than a space character is, then we consider this to be a label and extract it using the `copyText` code, coming soon.

Labels are copied from the input buffer to the label buffer.

```

;------------------------------------------------------------------------------
; Copy any opcode from the inputBuffer to the labelBuffer. A4 should
; be the first character in the inputBuffer. If the extracted opCode
; doesn't need an operand, we set that flag accordingly.
;
; This routine leaves A5 1 byte past the last character read if the
; opcode is NOT 3 or 5 in size - otherwise it will be the address of
; the 1st or 3rd character read, depending on the opcode. See below.
;------------------------------------------------------------------------------
extractOpcode
            lea     opcodeBuffer+2,a5   ; Output buffer
            bsr.s   doCopyText          ; Extract & copy opcode

;------------------------------------------------------------------------------
; A5 now points one past the last character copied. A1 is still
; pointing at the first character read. D0 is the size of the opcode.
; If the opcode is not 3 or 5 in size, it needs an operand.
; The noOperand flag is currently reset as per the start of readLoop.
;------------------------------------------------------------------------------
checkThree
            cmpi.w  #3,d0               ; Did we get three characters?
            beq.s   doThreeFive         ; Yes, skip

checkFive
            cmpi.w  #5,d0               ; Did we get 5 characters?
            beq.s   doThreeFive         ; Yes, skip

notThreeFive
            rts                         ; We need an operand

;------------------------------------------------------------------------------
; We get here if the opcode is 3 or 5 characters, now, is it one of
; the ones we want?
; We check the first 2 characters for 'no', 'rt', 're' or 'tr' and if
; found we have to check the remainder of the opcode to see if it is
; one which doesn't require an operand.
;
; These are: nop, reset, rte, rtr, rts, trapv.
```

```
;
; Reset and trapv are easy as they are the only 5 character opcodes
; starting with 're' or 'tr' and they both do not take operands.
;-------------------------------------------------------------------------------
doThreeFive
            move.l  a1,a5               ; Save first character start
            move.w  (a1),d1             ; Get first 2 characters
            ori.w   #lowerCase2,d1      ; Make lower case
            cmpi.w  #'no',d1            ; NO for NOP
            beq.s   doNO                ; Yes, skip
            cmpi.w  #'rt',d1            ; RT for RTE, RTR, RTS
            beq.s   doRT                ; Yes, skip
            cmpi.w  #'re',d1            ; RE for RESET
            beq.s   doTRRE              ; Yes, skip
            cmpi.w  #'tr',d1            ; TR for TRAPV
            bne.s   notThreeFive        ; No, exit

;-------------------------------------------------------------------------------
; This could be trapv or reset ... which as they are the only 5
; character opcodes that starts with 'tr' or 're' we must have a hit.
; Exit with A5 pointing at the 1st character of the opcode.
;-------------------------------------------------------------------------------
doTRRE
            bset    #noOperand,d5       ; There is no operand
            rts                         ; Done

;-------------------------------------------------------------------------------
; This could be rte, rtr, rts ...
; Exit with A5 pointing at the third character of the opcode.
;-------------------------------------------------------------------------------
doRT
            addq.l  #2,a5               ; Next two characters
            move.b  (a5),d1             ; Only need 1 character
            ori.b   #lowerCase1,d1      ; Make lower case
            cmpi.b  #'e',d1             ; RTE?
            beq.s   doTRRE              ; Yes
            cmpi.b  #'r',d1             ; RTR?
            beq.s   doTRRE              ; Yes
            cmpi.b  #'s',d1             ; RTS?
            beq.s   doTRRE              ; Yes
            rts                         ; It's not one of the above

;-------------------------------------------------------------------------------
; This could be nop ...
; Exit with A5 pointing at the third character of the opcode.
;-------------------------------------------------------------------------------
doNO
            addq.l  #2,a5               ; Next two characters
            move.b  (a5),d1             ; Only need 1 character
            ori.b   #lowerCase1,d1      ; Make lower case
            cmpi.b  #'p',d1             ; NOP?
            beq.s   doTRRE              ; Yes
            rts                         ; It's not NOP
```

Listing 3.17: ASMReformat Source - Extracting Opcodes

The subroutine above extracts opcodes from the input line. It does this by calling the `copyText` subroutine. Opcodes are copied from the input buffer to the opcode buffer.

Once an opcode has been extracted we check it to see if it requires an operand or not. Any operand which is not exactly three or five characters in size needs an operand.

If the length of the opcode is three or five, then we lower case the first two characters and begin a search for `NOP`, `RTE`, `RTR`, `RTS`, `RESET`, `TRAPV`. If those are found we set the `noOperand` flag so that we don't attempt to extract any operands for these instructions.

`RESET` and `TRAPV` are the only 5 character instructions, beginning with 'RE' and 'TR' that do not have operands, so if we have a 5 character opcode that begins with either of those two characters, then we definitely have to set the flag.

```

;------------------------------------------------------------------------------
; Copy any operand from the inputBuffer to the operandBuffer. If this
; opcode has no operands, do nothing, otherwise extract the operand
; into the buffer. A4 is the input buffer pointer.
; If the operand ends with a comma, then we need to set the continue
; flag for the next line to continue the operand.
;------------------------------------------------------------------------------
extractOperand
            btst    #noOperand,d5          ; Do we need to do anything?
            bne.s   extractOperandDone     ; No, skip

;------------------------------------------------------------------------------
; We have an operand, copy it to the operandBuffer. Keep a count of
; chars copied in D0.
;------------------------------------------------------------------------------
            bclr    #continue,d5           ; Assume no continuation
            lea     operandBuffer+2,a5     ; Our output buffer
            bsr     doCopyText             ; Copy operand
            cmpi.b  #comma,-1(a5)          ; Last character a comma?
            bne.s   extractOperandDone     ; No, skip
            bset    #continue,d5           ; We have a continuation

extractOperandDone
            rts
```

Listing 3.18: ASMReformat Source - Extracting Operands

The subroutine above extracts operands from the input line. It does this by calling the `copyText` subroutine. Operands are copied from the input buffer to the operand buffer.

We obviously do not need to extract an operand if the flag that says not to is set. After extracting the operand, if the final character was a comma (,) then we need to continue extracting this operand on the following line(s) of the input file, so we set the `continue` flag before continuing.

```

;------------------------------------------------------------------------------
; Copy any comment from the inputBuffer to the commentBuffer. A4 is
; the input buffer pointer. Returns with A5 one past the last char.
; Never returns here though.
;------------------------------------------------------------------------------
extractComment
            bset    #inComment,d5          ; We are doing comments
            lea     commentBuffer+2,a5     ; Our output buffer
```

```
                bra     doCopyText              ; Copy comment
```

Listing 3.19: ASMReformat Source - Extracting Comments

The subroutine above extracts comments from the input line. It does this by calling the `copyText` subroutine after setting a flag that indicates that we are in a comment. Comments are copied from the input buffer to the comment buffer.

The flag set indicates to the `copyText` code that all characters are valid, even spaces, tabs, etc - up to the terminating linefeed.

The hard work of extracting labels, opcodes etc is done by the code that follows.

```

;------------------------------------------------------------------------------
; Copy text from the input buffer (A4) to the output buffer (A5) and
; keep a count in D0. Scan forward in the input until we hit a non-
; space/tab character. Newline indicates the buffer end.
; A1 is a pointer to the start of the output buffer on entry and will
; be used to save the word count on completion.
; Watch out for quotes!
; If we are in a comment, then simply scan until the end.
;------------------------------------------------------------------------------
copyText
                bsr.s   scanForward             ; Locate next valid character
                moveq   #0,d0                   ; Counter

copyLoop
                cmpi.b  #linefeed,(a4)          ; Done yet?
                beq.s   copyTextDone            ; Yes, return
                btst    #inComment,d5           ; Are we in a comment?
                bne.s   copyComment             ; Yes, skip

;------------------------------------------------------------------------------
; We are not in a comment, so check for quotes. If we find one we
; must copy all characters until we get to the end quote. Otherwise
; any space/tab/newline character will end this copy.
;------------------------------------------------------------------------------
                cmpi.b  #sQuote,(a4)            ; Single quote?
                beq.s   copyString              ; Yes, skip
                cmpi.b  #dQuote,(a4)            ; Double quote?
                beq.s   copyString              ; Yes, skip

;------------------------------------------------------------------------------
; Not in a quoted string, are we done yet? If not, copy the current
; character and go around again.
;------------------------------------------------------------------------------
                cmpi.b  #space,(a4)             ; Done yet?
                bls.s   copyTextDone            ; Yes, return
                bra.s   copyComment             ; Copy one character

;------------------------------------------------------------------------------
; We have found a quote, grab it, then copy & scan to the end quote.
;------------------------------------------------------------------------------
copyString
                move.b  (a4)+,d1                ; Grab opening quote
                move.b  d1,(a5)+                ; Save opening quote
```

```
            addq.w  #1,d0                   ; Update counter

;------------------------------------------------------------------
; We have copied the start quote and incremented counters & pointers
; so we are now ready to copy the remaining characters in the quoted
; string.
;------------------------------------------------------------------
copyCharLoop
            move.b  (a4)+,(a5)              ; Copy current character
            addq.w  #1,d0                   ; Update counter
            cmp.b   (a5),d1                 ; Copied closing quote?
            addq.l  #1,a5                   ; Dest address, Z unchanged
            bne.s   copyCharLoop            ; No, keep copying
            bra.s   copyLoop                ; String done, carry on

;------------------------------------------------------------------
; If we are in a comment, we don't care what characters we read as
; all are required up to the terminating linefeed.
;------------------------------------------------------------------
copyComment
            move.b  (a4)+,(a5)+             ; Copy character
            addq.w  #1,d0                   ; Increment counter
            bra.s   copyLoop                ; Do some more

;------------------------------------------------------------------
; At the end, store the word count at the start of this buffer.
;------------------------------------------------------------------
copyTextDone
            move.w  d0,-2(a1)               ; Save text length
            rts
```

Listing 3.20: ASMReformat Source - Copying Input Source Lines Around

This subroutine reads the input and copies valid text to whatever buffer is pointed to by `A5`. On entry, the input pointer `A4` could be pointing anywhere in the input buffer, so in order to find a valid starting point, we call out to `scanForward` to ignore anything that is not a printable ASCII character. On return, `(A4)` is pointing at either a linefeed - indicating end of input, or at a valid printable character.

`D0` is used to count the characters in whichever field (label, opcode, operand, comment) that we are extracting.

If the current character is a linefeed, we are done and we store `D0` at the start of the output buffer - which `A1` is pointing to - and return to the caller.

Assuming we have not hit the end yet, we enter some convoluted code to make sure that what we extract is valid. If we are in a comment, any character is allowed, so we skip off to copy the current character from the input buffer to the output buffer that we are using just now. An easy case.

If we are not in a comment, we must check if we have one or other of the two quote characters as the current character. In this case we are about to copy a string so again, all characters are allowed until we come across the terminating quote character.

The code at `copyString` first copies the opening quote to the output buffer, and saves it in `D1` so that we can check for the end of the string.

Then we copy each of the following characters to the output buffer but note that we don't update

A5 using post increment addressing, like we do with A4. We check the character just copied for a closing quote which sets the Z flag accordingly, *then* we update A5 which we can do without affecting any of the flags. This allows us to make sure we copy over the closing quote and still be able to check for when we have finished copying a string value.

If we are not copying a string then we only allow printable characters. If we have a space, or lower, as the current input character, we are done and exit by storing D0 in the start of the output buffer.

```

;------------------------------------------------------------------------------
; Scan forward to the next non space/tab character. A newline is the
; end of the line and that will cause a return. Actually, we simply
; test for anything less than of equal to a space, other than a
; linefeed and keep incrementing until we get something else.
;
; Expects A4 to point into the current inputBuffer and exits with A4
; pointing at the next non-space/tab character, which might be a line
; feed.
;------------------------------------------------------------------------------
scanForward
            cmpi.b  #linefeed,(a4)      ; Newline?
            beq.s   scanDone            ; Yes, done

            cmpi.b  #space,(a4)         ; Space (or less)?
            bhi.s   scanDone            ; No, done

            addq.l  #1,a4               ; Increment currentPointer
            bra.s   scanForward         ; Keep scanning

scanDone
            rts                         ; Done. (A4) is the next char
```

Listing 3.21: ASMReformat Source - Scanning the Input Lines

This subroutine looks at the input characters and increments the input pointer register, A4, until we hit either a linefeed or any character higher than space in the ASCII chart.

```

;------------------------------------------------------------------------------
; We don't have a comment or blank, nor do we have an operand that has
; been continued over two (or more) lines, so we need to extract all
; the data from the input line.
;------------------------------------------------------------------------------
extractData
            bsr     extractLabel        ; Get any label
            bsr     extractOpcode       ; Get opcode - sets noOperand
            bsr     extractOperand      ; get Operand - sets continue
            bsr.s   extractComment      ; Get comments - sets inComment
            bra.s   doLabel             ; Go do label processing
```

Listing 3.22: ASMReformat Source - Main Extraction Control Code

The code above calls out to the four field extraction subroutines described above, then on return, skips to the output routines below where the reformatting takes place prior to writing out the newly reformatted line.

```

;------------------------------------------------------------------------------
```

```
; Some code to write out some text at the current position in the
; output file. On error, will exit via errorExit and never return.
; Assumes A0 has the correct channel ID and that A1 points to a QDOS
; string ready to be printed.
;------------------------------------------------------------------------
doWrite
            moveq   #io_sstrg,d0            ; Trap code
            move.w  (a1)+,d2                ; Word count

;------------------------------------------------------------------------
; Do a trap #3 and only return to the caller if it worked. Otherwise
; exit back to SuperBASIC with the error code.
;------------------------------------------------------------------------
doTrap3
            trap    #3                      ; Write the line/byte
            tst.l   d0                      ; Ok?
            bne     errorExit               ; No, bad stuff happened.
            rts                             ; Back to caller

doLineFeed
            moveq   #io_sbyte,d0            ; Send a single byte
            moveq   #linefeed,d1            ; Byte to send
            bra.s   doTrap3                 ; Do it
```

Listing 3.23: ASMReformat Source - Trap #3 Code

The code above is a small collection of `TRAP #3` routines to write the output buffers, send linefeeds etc.

```

;------------------------------------------------------------------------
; Copy a buffer from the word count at (A1) to the byte space at (A5)
; this is used when we copy the various buffers to the inputBuffer
; which we are using as an output Buffer now!
;
; Uses A1 as the source, A5 as the dest and D0.W as a counter.
; Corrupts A1 and D0.W. A5 exits as the next free byte in the buffer.
;------------------------------------------------------------------------
copyBuffer
            move.w  (a1)+,d0                ; Counter
            beq.s   copyBufferDone          ; Nothing to do, return
            subq.w  #1,d0                   ; Adjust for dbra

copyBufferByte
            move.b  (a1)+,(a5)+             ; Copy a byte
            dbra    d0,copyBufferByte       ; And the rest

copyBufferDone
            rts                             ; Back to caller
```

Listing 3.24: ASMReformat Source - Copying Buffers Around

The code above is called when we need to copy one of the input buffers holding labels, opcodes, operands or comments, back into the output buffer at the appropriate character position.

```

;------------------------------------------------------------------------
```

```
; Space fill the inputBuffer prior to using it as the outputBuffer to
; write the reformatted line to the output file.
;------------------------------------------------------------------------------
clearBuffer
            move.w   #255,d0                ; Counter for 256 longs
            lea      inputBuffer,a0         ; Guess!
            move.w   #0,(a0)+               ; No string in buffer

clearBufferLong
            move.l   #$20202020,(a0)+       ; Clear one long
            dbra     d0,clearBufferLong     ; Do the rest
            rts
```

Listing 3.25: ASMReformat Source - Clearing the Input Buffer

We use the input buffer for our output buffer too, so before we start, we need to make sure that it is space filled.

```

;------------------------------------------------------------------------------
; Labels get a line of their own, so we will write out the label by
; itself before looking at the rest of the stuff on the line.
;------------------------------------------------------------------------------
doLabel
            lea      labelBuffer,a1         ; Label word count
            tst.w    (a1)                   ; Any label?
            beq.s    doOpcode               ; No, skip
            move.l   destId(a7),a0          ; Destination channel id
            bsr.s    doWrite                ; Print out the label by itself
            bsr.s    doLineFeed             ; And a linefeed
```

Listing 3.26: ASMReformat Source - Writing Labels

If we have extracted a label from the input source line, we write it out here, by itself and follow it with a linefeed. Labels get written to the start of the output line, so we simply write the label out from the label buffer where it can currently be found.

```

;------------------------------------------------------------------------------
; If we have an opcode, and normally we should have one, tab to the
; desired position and write it out. It is not normal for an opcode
; to exceed the space allocated, so no checks are done here.
; We always clear the inputBuffer at this point.
; We use D5 from here on to show if we printed anything and if so, we
; will need a linefeed afterwards, otherwise, no linefeed is needed.
;------------------------------------------------------------------------------
doOpcode
            bsr.s    clearBuffer            ; We always do this here
            bclr     #lfRequired,d5         ; Nothing printed so far
            lea      opcodeBuffer,a1        ; Source word count
            tst.w    (a1)                   ; Got an opcode?
            beq.s    doComment              ; No opcode, no operand
            bset     #lfRequired,d5         ; Flag something (to be) printed
            lea      inputBuffer+opcodePos+2,a5   ; Dest byte area
            bsr.s    copyBuffer             ; Copy the opCode
```

Listing 3.27: ASMReformat Source - Writing Opcodes

Opcodes, if we have one, are copied from the opcode buffer to the input buffer at the desired position, after clearing the input buffer of its current contents. If the opcode has no operands, we skip down to checking for comments. After sending something to the output buffer, we set a flag to show that we must print a linefeed when done.

```

;------------------------------------------------------------------------------
; Write out an operand. This may be a new one, or a continuation. If
; the operand exceeds commentPos-2 then add a couple of spaces to the
; output line before the comment gets printed. We use D6.W to hold
; any extra bytes used for use below.
;
; If commentPos = 40 and operandPos = 20 then max operand size is
; 40 - 20 - 1 = 19 before we have to extend the comment position.
;------------------------------------------------------------------------------
doOperand
            moveq    #0,d6                    ; Extra byte counter
            lea      operandBuffer,a1         ; Operand word count
            tst.w    (a1)                     ; Do we have an operand?
            beq.s    doComment                ; No, skip
            bset     #lfRequired,d5           ; Something printed
            move.l   a1,a4                    ; Save buffer address
            lea      inputBuffer+operandPos+2,a5
            bsr.s    copyBuffer               ; Copy operand
            move.w   (a4),d0                  ; Operand size
            cmpi.w   #commentPos-operandPos-1,d0  ; Check width
            bls.s    doComment                ; Narrow operand

doWideOperand
            addq.l   #2,a5                    ; Adjust A5
            moveq    #2,d6                    ; Two extra bytes now
```

Listing 3.28: ASMReformat Source - Writing Operands

Operands, if we have one, get copied into the output buffer at the desired location. There are no opcodes that are so long that they span from the opcode position over the operand position, so no checks are done here.

If you decide to change the buffer positions then you might need to do some additional checks here. Caveat emptor and all that!

Operands, on the other hand, *might* span well past the comment position, so, if they do, we add a couple of extra spaces to the output and set D6 to show that we have a wide operand to cope with.

```

;------------------------------------------------------------------------------
; If we have a comment then print it at the desired position. If the
; operand took too much space (above) then offset the comment by a
; couple of extra spaces - as per D6.W.
; If there is no comment, then simply print a linefeed, if required.
;------------------------------------------------------------------------------
doComment
            lea      commentBuffer,a1         ; Comment word count
            tst.w    (a1)                     ; Do we have a comment?
            beq.s    addLineFeed              ; No, skip
            bset     #lfRequired,d5           ; Something printed

```

```
;------------------------------------------------------------------------
; If D6 is non-zero, then A5 is set to the correct output byte,
; otherwise, set A5 to the normal comment position in the buffer.
;------------------------------------------------------------------------

            tst.w   d6                      ; Zero = normal comment position
            bne.s   commentPositionSet      ; Non-zero = A5 is set correctly

setNormalOperandComment
            lea     inputBuffer+commentPos+2,a5   ; Destination

commentPositionSet
            bsr     copyBuffer              ; Copy the comment
```

Listing 3.29: ASMReformat Source - Writing Comments

Comments either get printed at their desired position, or, if the operand was a wide one, wherever they happen to find space on the output line. If `D6` is zero, the desired position can be used, otherwise, `A5` is already set to the first available space on the output.

The comment is copied from the comment buffer to wherever `A5` is pointing to in the output buffer.

```

addLineFeed
            btst    #lfRequired,d5          ; Anything printed?
            beq     readLoop                ; No, read next input line
            move.b  #linefeed,(a5)          ; Tag on a linefeed

;------------------------------------------------------------------------
; By here A5 is the linefeed charater written to the buffer so we
; can get the size of the text now, quite easily. The word count is
; the lastCharacter (in A5) minus the bufferStart (in A1) minus 1.
;
; For example:
;
;     A1---> 012345
;              __NOPx <---A5
;
; x = LineFeed.
; _ = Unknown/don't care.
;
; We need to print 4 characters 'NOP' plus linefeed, so 5-0-1 = 4.
;------------------------------------------------------------------------
            lea     inputBuffer,a1          ; Buffer word count
            move.l  a5,d0                   ; Copy
            subq.l  #1,d0                   ; Minus 1
            sub.l   a1,d0                   ; Offset into buffer
            move.w  d0,(a1)                 ; Store in buffer
```

Listing 3.30: ASMReformat Source - End of Line Feed

If we have an opcode, operand and/or comment in the buffer, we need to print a linefeed after writing the line out, so if the flag is set, we append a linefeed to the output buffer.

We also have to determine how wide the output buffer is, so, we do this by calculating $A5 - A1 - 1$ and storing the result in the start of the buffer. The input buffer, now being used for output, is ready to be printed.

```

;------------------------------------------------------------------------
; Write the reformatted line to the output channel using the code in
; doWriteComment above. This also returns to the start of readLoop.
;------------------------------------------------------------------------
doWriteLine
            bra       doWriteComment        ; Print the line & loop around
```

Listing 3.31: ASMReformat Source - End of Main Loop

This short piece of code prints the output buffer and skips back to the top of the main loop, ready to process the next line from the input file.

```

;------------------------------------------------------------------------
; We have hit an error so we copy the code to D3 then exit via a
; forcible removal of this job. EXEC_W/EW will display the error in
; SuperBASIC, but EXEC/EX will not.
;------------------------------------------------------------------------
allDone
            moveq     #0,d0

errorExit
            move.l    d0,d3                 ; Error code we want to return

;------------------------------------------------------------------------
; Kill myself when an error was detected, or at EOF.
;------------------------------------------------------------------------
suicide
            moveq     #mt_frjob,d0          ; This job will die soon
            moveq     #me,d1
            trap      #1
```

Listing 3.32: ASMReformat Source - End of Job Code

When we hit EOF on the input file, we exit the main loop and arrive at `allDone` where we flag no errors, and then the job kills itself.

Had we hit any errors in processing the input or output files, the main loop would exit via `errorExit` where we set `D3` as required by QDOSMSQ, and then kill the job.

Assuming we executed the utility using `EW` we would be able to see the error message.

```

;------------------------------------------------------------------------
; Various buffers. Having them here keeps them separate from code and
; makes it easier for disassemblers to decode the code without having
; to worry about embedded data!
;------------------------------------------------------------------------

inputBuffer
            ds.w      512+1                 ; Input - 1024 bytes + count

labelBuffer
            ds.w      128+1                 ; Label - 256 bytes + count

opcodeBuffer
            ds.w      10+1                  ; Opcode - 20 bytes + count
```

```

operandBuffer
             ds.w     128+1                    ; Operand - 256 bytes + count

commentBuffer
             ds.w     246+1                    ; Comment - 492 bytes + count
```

Listing 3.33: ASMReformat Source - Various Buffers

And finally - you will be glad to hear - these are the various buffers used by the utility to store the input (and output) as well as temporary storage for the 4 separate fields in an assembly source line.

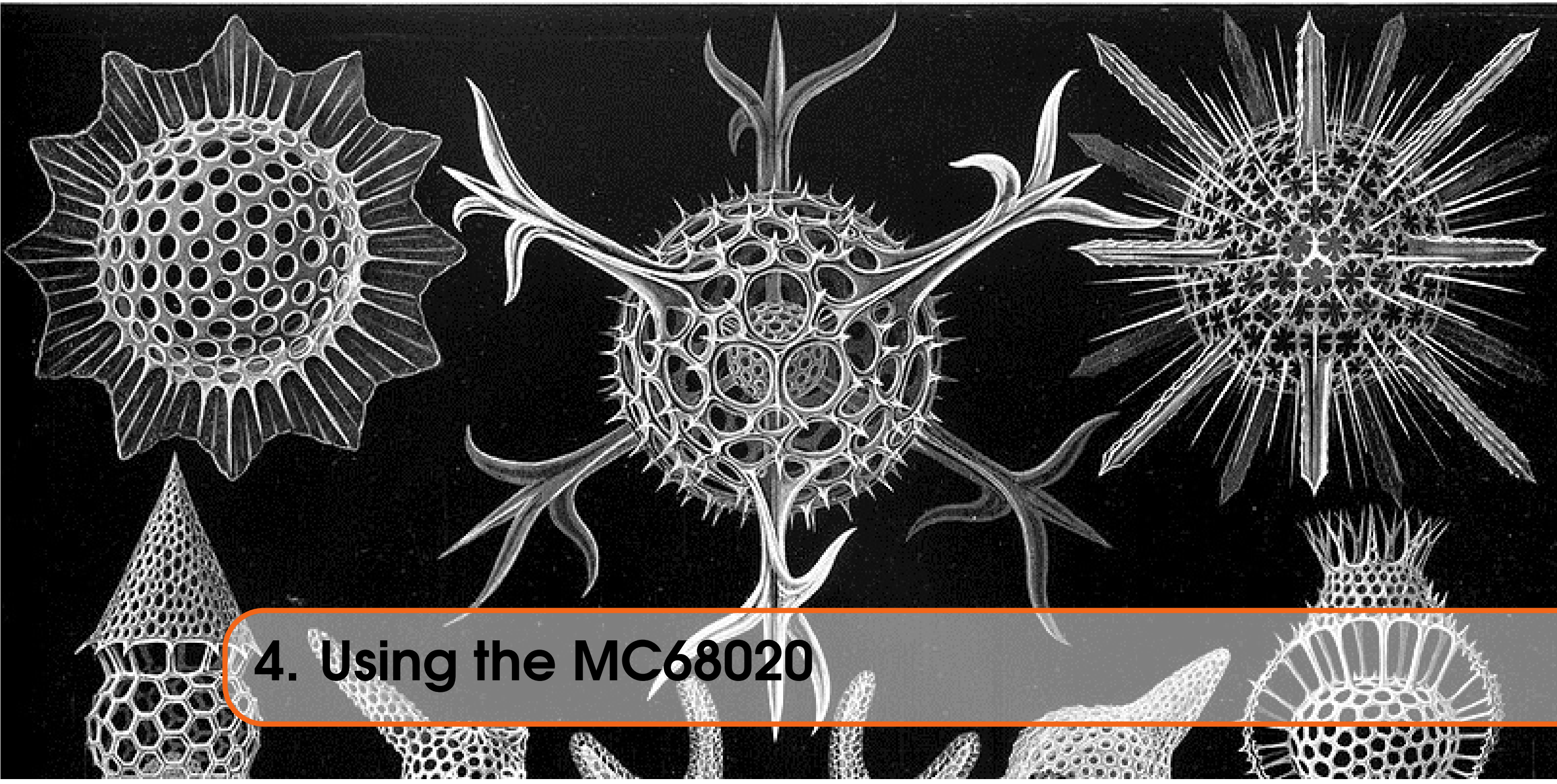
3.2 Finally

Are you wondering about some of the labels used in the above source code? Do they look odd? or simply too long etc? Well, the reason for that is that this code was used to reformat itself, so I was testing with short and long labels and found too many problems, so I ended up just putting labels on an output line by themselves.

Hopefully you will find this utility useful. I know I have done - so far!

One last thing, on Linux, the C++ version of this utility compiled down to around 38 KB - which is not much for a useful utility. However, it does use a number of shared libraries to carry out a lot of the hard work and those are not included in the 38 Kb.

The entire utility on my QPC setup, assembled to a total of 2,728 Bytes!



4. Using the MC68020

As mentioned in the last issue, I am planning on upgrading the eComic to use the 68020 instructions available in QPC and in George's Gwass assembler. This currently means that unless you have a Q40 or Q60 to hand, you will need to run the programs and assembler on QPC. Is this a problem I wonder?

4.1 Overview

Here are a few brief details of what a proper 68020 has to offer:

- Full support for 32 bit operations.
- A full 32 bit external data bus which can also cope with 16 or 8 bit peripherals.
- 32 bit offsets in branch instructions.
- 32 bit displacements in indexed addressing modes.
- Two new addressing modes are provided, which allow indexed address with *two* levels of indirection.
- Word and Long memory accessing need no longer be on an even address.
- New bit field instructions.
- Instructions to convert between character and decimal numbers.
- And lots, lots more!

4.2 Addressing Modes

The addressing modes of the 68008 are mostly familiar, or should be by now, however, here is a reminder of those modes, plus the new modes available in the 68020. The mode number given is that coded into the mode bits of the effective address in the various instructions. (But you don't really *need* to know this!)

In the following descriptions, I've taken the wording as is from the Motorola Programmers' Manual

- hence the strangeness of some of the wording. The examples, however, are mine.

4.2.1 Data Register Direct

Mode zero. The effective address specifies the data register that the contains the operand. For example `move.w #1,d0` the destination address is Data Register Direct.

4.2.2 Address Register Direct

Mode 1. The effective address specifies the address register that the contains the operand. For example `move.l $c0ffee,a3`. The destination effective address is `A3` and is indeed Address Register Direct.

4.2.3 Address Register Indirect

Mode 2. The effective address specifies the address register that the contains the operand in memory. For example `move.w (a3),d7` where `A3` holds the address where the operand, a word of data, is to be found.

4.2.4 Address Register Indirect with Post-Increment

Mode 3. In the address register indirect with postincrement mode, the operand is in memory. The effective address field specifies the address register containing the address of the operand in memory.

After the operand address is used, it is incremented by one, two, or four depending on the size of the operand: byte, word, or long word, respectively.

For example `move.w (a0)+,d0` will read the word value from the address held in `A0` into `D0` and then will add two - the size of a word - to A3.

If the address register is `a7`, the stack pointer, then byte sized operations cause `a7` to be incremented by 2, rather than by 1. For example `move.b d0, (a7)+` will cause `A7` to be incremented by two to keep it even.

The address register in question retains the new incremented value after the instruction.

4.2.5 Address Register Indirect with Pre-Decrement

Mode 4. In the address register indirect with predecrement mode, the operand is in memory. The effective address field specifies the address register containing the address of the operand in memory.

Before the operand address is used, it is decremented by one, two, or four depending on the operand size: byte, word, or long word, respectively.

For example `move.l -(a0),d0` causes `A0` to be decremented by 4 and the long word found at the new address will be moved into `D0`.

If the register is `A7`, the stack pointer, then byte sized operations cause `A7` to be incremented by 2, rather than by 1. For example `move.b d0,-(a7)`.

The address register in question retains the new decremented value after the instruction.

4.2.6 Address Register Indirect with Displacement

Mode 5. In the address register indirect with displacement mode, the operand is in memory.

The sum of the address in the address register, which the effective address specifies, plus the sign-extended 16-bit displacement integer in the extension word is the operand's address in memory.

Displacements are always sign-extended to 32 bits prior to being used in effective address calculations.

For example `move.l d0,$10(a1)` will sign-extend the 16 bit displacement word (10_{hex}) to a full 32 bit signed value, add it to the address held currently in `A1` - without affecting the actual address held in the register - and the long value in `D0` will then be stored there.

The displacement can of course, be negative, `move.l d0,-$14(a1)`.

The displacement word is 16 bits, however, it will always be sign extended to 32 bits prior to the addition to the address register.

The address register in question retains its *current* value after the instruction - it is not adjusted in any way.

4.2.7 Address Register Indirect with Index (8 bit Displacement)

Mode 6. This addressing mode requires one extension word that contains an index register indicator and an 8-bit displacement. The index register indicator includes size and scale information.

In this mode, the operand is in memory. The operand's address is the sum of the address register's contents; the sign-extended displacement value in the extension word's low-order eight bits; and the index register's sign-extended contents (possibly scaled).

The user *must* specify the address register, the displacement, and the index register in this mode - none of these are optional, only the scaling factor is optional and will default to 1 if omitted.

For example `move.w 4(a6,d7.W),d3`. In this example, the 8 bit displacement value, 4, is sign extended to 32 bits and added to the address held in `A6`. The value in `D7` is also sign extended to 32 bits and added to the above calculation. The word value at this calculated address is copied into `D3`.

The calculated address is not stored anywhere, it is used and discarded. The value in the address register, `A6` in this case, is not affected.

The index register, `D7` may, optionally, have its value scaled - which the example code shown below attempts to explain.

It seems,according to the Programmers' manual, that we *should* be writing the above example as `move.w (4,a6,d7.W),d3` instead. Luckily GWASS is happy with the old style[1] as well as the new.

As mentioned, both the 8 bit displacement and the index register, if word sized, will be sign extended to 32 bits before being used in effective address calculations.

As for the scaling mode mentioned above, do you remember last issue's jump tables article? Well, here's a reminder[2]:

```
got_good_option
        subq.b #'0',d0          ; D0.B = 0 to 9 as a number
```

[1] My preferred style!

[2] Corrected as per George's comments!

```
        ext.w d0                ; Now extend to a word
        lsl.w #1,d0             ; Convert to a table offset
        lea JumpTable,a2        ; Where the jump table lives
        move.w 0(a2,d0.w),d0    ; Fetch the offset word
        jsr (a2,d0.w)           ; Jump to the correct subroutine
```

Listing 4.1: Jump Table - Old Style

Now, with the 68020 and scaling, there's no need to do the separate doubling of the table's index (`lsl.w #1,d0`) to calculate the correct offset into the table as the scaling does this automatically and *without changing* `d0`. The above extract would be as written as follows:

```
got_good_option
        subq.b #'0',d0          ; D0.B = 0 to 9 as a number
        lea JumpTable,a2        ; Where the jump table lives
        move.w 0(a2,d0.w*2),d0  ; Fetch the offset word
        jsr (a2,d0.w)           ; Jump to the correct subroutine
```

Listing 4.2: Jump Table - New Style

You will notice that I have specified the displacement (0) and both the address (`A2`) and index register (`D0.W`) as required.

4.2.8 Address Register Indirect with Index (Base Displacement)

Also mode 6. This addressing mode requires an index register indicator and an optional 16- or 32-bit sign-extended base displacement. The index register indicator includes size and scaling information. The operand is in memory.

The operand's address is the sum of the contents of the address register, the base displacement, signed extended if necessary, and the scaled contents of the sign-extended index register.

In this mode, the address register, the index register, and the displacement are *all optional*.

The effective address is zero if there is no specification. This mode can provide a data register indirect address when there is no specific address register and the index register is a data register.

The example for this addressing mode is similar to the one above, however you don't need to specify all of the fields and scaling. For example we can change our addressing mode from `move.w 0(a2,d0.w*2),d0` to `move.w (a2,d0.w*2),d0` where the displacement is optional.

As mentioned, this mode can give you a pseudo *Data register Indirect* addressing mode, simply by leaving off most of the optional fields. For example, under the 68020, the following is valid `move.w (d0.l),d0` - assuming that `D0.L` contains a valid 'address' of course.

4.2.9 Memory Indirect Postindexed

Also mode 6. In this mode, both the operand and its address are in memory. The processor calculates an intermediate indirect memory address using a base address register and base displacement.

The processor accesses a long word at this address and adds the index operand (Xn.SIZE*SCALE) and the outer displacement to yield the effective address.

Both displacements and the index register contents are sign-extended to 32 bits.

In the syntax for this mode, square brackets [] enclose the values used to calculate the intermediate memory address.

All four user-specified values are optional.

Both the base and outer displacements may be null, word, or long word. When omitting a displacement or suppressing an element, its value is zero in the effective address calculation.

For example `move.l ([8,a6],d4.w*4,96),d0` will calculate a temporary address in memory by adding the sign-extended base displacement (8) and the address register (`A6`). This address will contain a long word which is read, added to the sign-extended index register (`D4.W*4`), plus the outer displacement (96). Phew!

The immediate question that comes to my mind is “why?” However, there must have been a reason for this addressing mode to be built in silicon.

4.2.10 Memory Indirect Preindexed

Mode 6 again. In this mode, both the operand and its address are in memory. The processor calculates an intermediate indirect memory address using a base address register, a base displacement, and the index operand (Xn.SIZE*SCALE).

The processor accesses a long word at this address and then adds the outer displacement to yield the effective address.

Both displacements and the index register contents are sign-extended to 32 bits.

In the syntax for this mode, brackets enclose the values used to calculate the intermediate memory address.

All four user-specified values are optional.

Both the base and outer displacements may be null, word, or long word. When omitting a displacement or suppressing an element, its value is zero in the effective address calculation.

For example `move.l ([18,a5,d4.w*4],200),d0` will calculate a temporary address in memory by adding the sign-extended base displacement (18), the address register (`A5`) and the sign-extended index register (`D4.W*4`). The long word at that address will then be read and added to the outer displacement (200) and whatever long word is found at that address will be copied into `D0`. Phew!

The immediate question that again comes to my mind is “why?”

4.2.11 Absolute Short

Mode 7 Submode 0. In this addressing mode, the operand is in memory, and the address of the operand is in the extension word. The 16-bit address is sign-extended to 32 bits before it is used.

For example `move.w $1234,d4` takes 2 words of memory. The first defines the opcode and the second word defines the short address. The second word is read, sign-extended to 32 bits and the word, in this example, at that address is copied into `D4`.

Note that addresses between \$0000 and \$7fff sign-extend to the same values, but addresses from \$8000 to \$ffff sign-extend to actual addresses of \$ffff8000 to \$ffffffff. So, effectively, you can only use this addressing mode on the lowest 32KB of memory and, if you have enough RAM, the upper 32KB of memory.

4.2.12 Absolute Long

Mode 7 Submode 1. In this addressing mode, the operand is in memory, and the operand's address occupies the two extension words following the instruction word in memory.

The first extension word contains the high-order part of the address; the second contains the low-order part of the address.

For example `move.b $12345678,d4` takes 3 words of memory. The first defines the opcode, the second word defines the high half of the address $1234 and, finally, the third word defines the low half of the address $5678. The two words are read, and the byte, in this example, at that address is copied into `D4`.

4.2.13 Program Counter Indirect with Displacement

Mode 7 Submode 2. In this mode, the operand is in memory. The address of the operand is the sum of the address in the program counter (PC) and the sign-extended 16-bit displacement integer in the extension word. The '(PC)' part of the opcode can be left off as it is optional.

The value in the PC is the address of the extension word defining the offset.

This is a program reference allowed only for reads.

For example, `lea jumptable(pc),a2` will set `A2` to the position independent location of the label `jumptable` no matter which address in RAM the code is running at. In memory, there are two words. The first defines the opcode, the second, which is where the Program Counter is pointing, is the displacement to the given label from the current address of the PC.

The example could also have been written as `lea jumptable,a2`

4.2.14 Program Counter Indirect with Index (8-Bit Displacement)

Mode 7 Submode 3. This mode is similar to the mode described in *Address Register Indirect with Index (8 bit Displacement)* on page 43 , except the PC is the base register.

The operand is in memory.

The operand's address is the sum of the address in the PC, the sign-extended displacement integer in the extension word's lower eight bits, and the sized, scaled, and sign-extended index operand.

The value in the PC is the address of the extension word.

This is a program reference allowed only for reads.

The user *must* include the displacement, the PC, and the index register when specifying this addressing mode.

For example `move.w jumptable(pc,d0.w*2),d0` could have been used in our jump table example above as it does not require the use of a base register to access the table to fetch the offset.

4.2.15 Program Counter Indirect with Index (Base Displacement)

Mode 7 Submode 3 again. This mode is similar to the mode described in *Address Register Indirect with Index (Base Displacement)* on page 44, except the PC is used as the base register.

It requires an index register indicator and an optional 16 or 32 bit sign-extended base displacement.

The operand is in memory.

The operand's address is the sum of the contents of the PC, the base displacement, and the scaled contents of the sign-extended index register.

The value of the PC is the address of the first extension word.

This is a program reference allowed only for reads.

For example `lea jumptable(PC,d0.w*2),a3` will work out the address of the `D0`th word in the table at label `jumptable` and copy it into `A3`.

In this mode, the PC, the displacement, and the index register are optional. The user must supply the assembler notation ZPC (a zero value PC) to show that the PC is not used. This allows the user to access the program space without using the PC in calculating the effective address.

The user can access the program space with a data register indirect access by placing ZPC in the instruction and specifying a data register as the index register.

I have to admit that I'm not convinced that a PC or zero is going to be useful, certainly not in program independent code.

4.2.16 Program Counter Memory Indirect Postindexed Mode

Mode 7 Submode 3 also. This mode is similar to the mode described in *Memory Indirect Postindexed* on page 44, but the PC is the base register.

Both the operand and operand address are in memory.

The processor calculates an intermediate indirect memory address by adding a base displacement to the PC contents. The processor accesses a long word at that address and adds the scaled contents of the index register and the optional outer displacement to yield the effective address.

The value of the PC used in the calculation is the address of the first extension word.

This is a program reference allowed only for reads.

In the syntax for this mode, brackets enclose the values used to calculate the intermediate memory address. All four user-specified values are optional.

The user must supply the assembler notation ZPC (a zero value PC) to show the PC is not used. This allows the user to access the program space without using the PC in calculating the effective address.

Both the base and outer displacements may be null, word, or long word. When omitting a displacement or suppressing an element, its value is zero in the effective address calculation.

For an example, see *Memory Indirect Postindexed* on page 44 and replace the address register with 'PC'.

4.2.17 Program Counter Memory Indirect Preindexed Mode

Mode 7 Submode 3 also again! This mode is similar to the mode described in *Memory Indirect Preindexed* on 45, but the PC is the base register.

Both the operand and operand address are in memory.

The processor calculates an intermediate indirect memory address by adding the PC contents, a base displacement, and the scaled contents of an index register. The processor accesses a long word at immediate indirect memory address and adds the optional outer displacement to yield the effective address.

The value of the PC is the address of the first extension word.

This is a program reference allowed only for reads.

In the syntax for this mode, brackets enclose the values used to calculate the intermediate memory address. All four user-specified values are optional. The user must supply the assembler notation ZPC showing that the PC is not used.

This allows the user to access the program space without using the PC in calculating the effective address.

Both the base and outer displacements may be null, word, or long word. When omitting a displacement or suppressing an element, its value is zero in the effective address calculation.

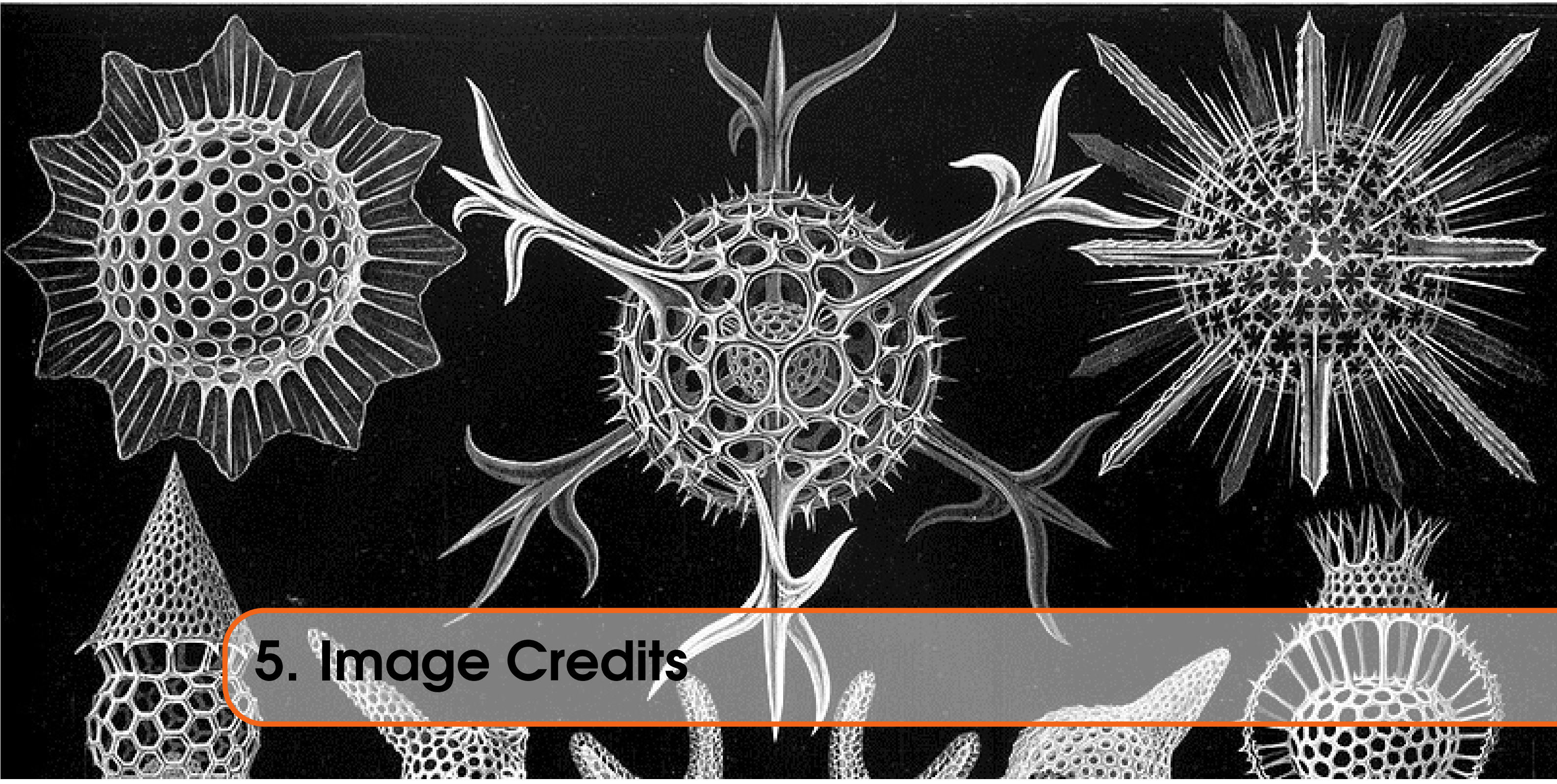
For an example, see *Memory Indirect Preindexed* on page 45 above and replace the address register with 'PC'.

4.2.18 Immediate Data

Mode 7 Submode 4. In this addressing mode, the operand is in one or two extension words.

For example, `move.l #100,d0`. After this instruction has executed, `D0` will contain the value $100_{decimal}$ in all 32 bits.

That's it for the complete set of addressing modes. Next time, our exploration of the 68020 instructions will take a good look at the various Bit Field Instructions.



5. Image Credits

The front cover image on this ePeriodical is taken from the book *Kunstformen der Natur* by German biologist Ernst Haeckel. The book was published between 1899 and 1904. The image used is of various *Polycystines* which are a specific kind of micro-fossil.

I have also cropped the image for use on each chapter heading page.

You can read about Polycystines on Wikipedia and there is a brief overview of the above book, also on Wikipedia, which shows a number of other images taken from the book. (Some of which I considered before choosing the current one!)

Polycystines have absolutely nothing to do with the QL or computing in general - in fact, I suspect they died out before electricity was invented - but I liked the image, and decided that it would make a good cover for the book and a decent enough chapter heading image too.

Not that I am suggesting, *in any way whatsoever*, that we QL fans are ancient.